Network Anomaly Detection With Graph Neural Networks

Group 21

Emily Hannon

Nicholas Lannon

Gustavo Nazario Perez

Santiago Rodriguez

Landon Russell

Mukundh Vasudevan

Sponsor: Branden Stone (Georgia Tech Research Institute)

Table of Contents

1. Executive Summary

Beginning this project, our goal was to use Graph Neural Networks (GNNs) to detect anomalies in cybersecurity power grid scenarios and to create a new GNN that detects graph anomalies more accurately and/or efficiently by the end of our project. By the end of the project, we successfully created a GraphSAGE GNN that successfully detects some common cyberattacks and their categories.

A GNN is a Graph-based neural network that takes a graph as input and outputs some anomaly detection about the graph. "Graph anomalies are patterns in a graph that do not conform to normal patterns expected of the attributes and/or structure of the graph." [1] The detection of anomalies for power grid scenarios is crucial as it ensures the reliability and security of critical infrastructure.

Power grid networks have many intricate relationships that can make them challenging to model. For example, fault propagation requires us to be certain of all connections between components and how one component can destroy the entire grid. Due to there being many complex interactions, traditional methods of anomaly detection, such as rule or clustering-based detection, fall short when trying to complete these tasks.

We set up a preliminary environment for running a simulation from the Network Attack Testbed In [Power] Grid (NATI[P]G) [2], and it is our hope that the final product will be tested on the fully on NATI[P]G data by future teams once the simulation has been fully developed. However, our initial tests took place on the older UNSW-NB15 Dataset [3], which is in the form of CSV files and can be downloaded and experimented on more quickly. The

hope is to develop technology that will improve Intrusion Detection Systems for distribution power grid networks by leveraging the networks' graph structure through the use of GNNs. This will open up doors for more advanced and adaptive anomaly detection systems, specifically those using Graph Neural Networks.

2. Introduction

2.1 Project Identification

Our team developed a graph neural network based on an extension of the current cutting-edge. As stated in the summary, our sponsor has given us power grid data to work with in order to detect anomalies in cyberattack scenarios. Improvements in this area will be extremely helpful. As current technology improves and advances, the United States power grid has become more susceptible to digital attacks with the advent of the Smart Grid.

According to the U.S. government, there are several advantages to developing a smart power grid which include: "more efficient transfer of electricity," "quicker restoration of electricity after power disturbances," "reduced operations and management costs for utilities, and ultimately lower power costs for consumers," "reduced peak demand, which will also help lower electricity rates," "increased integration of large-scale renewable energy systems," "better integration of customer-owner power generation systems, including renewable energy systems," "improved security." [4] Unfortunately, the implementation of a computerized power grid exposes our power grid to the possibility of cyberattacks from vandals or even outside organizations, threatening national security and safety.

As a team, we are motivated to improve cyberattack detection, specifically in this area, through the use of a Graph Neural Network. This neural network structure leverages the graph structure inherent to the smart power grid itself. As engineers and computer scientists, we also find the topic of Graph Neural Networks themselves interesting. Before beginning this

project, most of the team had never heard of a Graph Neural Network, so our own intellectual curiosity in the technology and its potential to be applied in different scenarios is another driving motivator for our group.

2.2 Project Significance

Digital communication networks have become integral in supporting daily life in the 21st Century. The United States especially depends on these to provide vital services to the nation, including our smart power grid. Consequently, digital communication networks have become increasingly relevant to national security. To prevent catastrophic failures, anomalies (like those caused by DDoS, injection, and parameter change attacks) must be detected quickly and with adequate description for repair. This project aims to solve this problem by leveraging the power of graph neural networks to take advantage of the graph structure inherent in digital communication networks.

According to a 2016 report from the Idaho National Laboratory, there were already many existing risks and vulnerabilities for cyber attacks on the U.S. electric sector. [5] Some threat actors outlined in this report included rogue individuals, hacktivists, terrorist groups, and countries such as Russia, China, Iran, and North Korea. Attacks may threaten three types of energy areas: production, transmission, and distribution. Our group can only assume that this threat has increased as more of our power grid systems have new computerized technology implemented for the previously outlined purposes.

According to the U.S. Energy Information, around 72% of electric utilities have "advanced (smart) metering infrastructure (AMI) installations." This number is slightly higher at 73% for residential meters, which vastly outnumber the three other categories, which are commercial, industrial, and transportation. [6] Since 2013, over double the amount of households have adopted the advanced metering infrastructure (AIM). [7] As we will later discuss, in a worst-case scenario, a successful cyberattack on a U.S. power grid can result in death, and such a scenario must be avoided at all costs.

Any improvements in cybersecurity will protect citizens from attacks originating both at home and abroad. Our group takes these threats very seriously, and they are one of the driving motivators for our group. While Graph Neural Networks are interesting in their own right, our project gains new meaning through the application of this technology to a smart power grid cyberattack scenario.

2.3 Individual Motivations and Ideas

2.3.1 Emily Hannon

2.3.1.1 Motivations

Personally, I am motivated to learn more about graph neural networks and cybersecurity in general. I have a lot of experience in applied research in other areas of machine learning, but I am looking forward to learning more about these topics. I am currently performing research in multiple applied artificial intelligence areas. One of my research projects is focused on Genetic Algorithms, which is a survival-of-the-fittest machine learning approach that copies the process of evolution in real-life biology.

During this project, we are looking at the application of a Genetic Algorithm to predict the location of G-Quadruplex structures with the aim of locating promoters and detecting transcription start sites in DNA using regular expression motifs.

I am also working on a thesis with the SENSEable Design Lab, which focuses on emotion prediction and team evaluation using multimodal input and machine learning. Additionally, my research group abroad works with the CERN ATLAS project to apply neural networks to create a more accurate reconstruction of the mass of the Higgs-Boson particle. Recently, we have been experimenting with the use of Population-Based Training [8], a hyperparameter tuning algorithm first posed by Deepmind that uses a genetic algorithm approach to training neural networks. While interesting from a physics perspective, this project is also interesting from a computer science perspective since there is little diversity in our data, making it extremely easy to overfit on.

I think this project will be interesting because it touches on topics I know very well and also topics that I still have a lot to learn about. I do not have much experience in the cybersecurity area, so it is going to be an area that I will be able to learn and grow in during the duration of this project. While I have a good amount of experience working on research projects in the AI and machine learning areas, I do not have any experience working with graph neural networks specifically. Overall, I am familiar with scientific and research methodologies that I believe will lead our team to success. I am also motivated on the research side of things to have a published paper, if the opportunity arises.

2.3.1.2 Ideas

In terms of where the project is headed, we should start with a literature review to help us brainstorm. During this process, we will be able to gather and discuss ideas as a group. Based on our first meeting with our sponsor and research mentor, our goal is to find the best and most cutting-edge graph neural network. Our sponsor has recommended multiple resources, including Papers With Code and Google Scholar. I have also suggested using arXiv in order to make the most of our literature review process.

Our sponsor has also suggested one paper as a starting point that gives a bit of an overview of the field of graph anomaly detection using graph neural networks as it stands today. [1] Looking at the papers that cite this and the papers that it cites will be a great starting point to read up on this field. We have also been provided with a data simulator, so this would be a great time to get that code ready and begin to run it. It would also be prudent to look at the second data set we have been provided with (which we will not be testing on) in order to get an idea of the data we are working with since it will be similar to our result from the simulation.

Once we have completed our literature review process, we need to brainstorm any extensions to this paper that can improve the neural network. During this process, we will need to plan our own neural network in detail and come up with an outline of what we expect to accomplish for the next semester. This includes code to be written and what experiments we plan to perform and run. Based on this planning, we may also need to request additional hardware to run our experiments on after we complete the coding.

After these steps, we can begin the implementation of the neural network and testing. At the end of the senior design project, we should have a report or paper written about our final results. As a stretch goal, it would be a fantastic opportunity for our entire team to turn our project into a published paper or part of a paper with the help of our sponsor.

2.3.2 Gustavo Nazario Perez

2.3.2.1 Motivations

This project piqued my interest due to combining machine learning with graph theory. I have taken courses on both of those topics individually at UCF and am looking forward to seeing how these concepts can be connected with Graph Neural Networks. As I work on this project with my team, I want to become more familiar with the math and processes of neural networks. Though I currently understand most of the ideas related to neural networks, I want to improve my clarity and remove any black boxes that still exist in my head. I believe that GNNs will be useful for this as I will be expanding upon my knowledge to learn something more complex, which will require me to have a strong understanding of the basics.

After talking with our sponsor, I found even more reasons to be excited to work on this project. A large process of our work is going to be performing literature reviews and becoming familiar with the state-of-the-art GNNs, specifically the ones used for anomaly detection. This is a skill that I find essential to any computer scientist, and this project enables us to use it. With technology rapidly changing, it is important to stay on top of discoveries and new ideas.

One of my personal goals in terms of my career is to work for a company that does work in the biotech industry. I find that it makes my work feel more impactful and is an interesting blend of computer science and biology. Learning about GNNs and how they can be used in anomaly detection is something that I believe is transferable to a career in the biotech industry.

Another essential part of this project is the understanding of networks and security. I do not have any aspirations of working in the cybersecurity field, but that is mainly because of my lack of exposure to it. I am hoping that this project will give me a better grasp of what cybersecurity is and how machine learning can be incorporated to increase communication safety amongst networks. UCF puts a strong emphasis on cybersecurity, likely because many companies in surrounding areas are in the defense industry, so knowing that field can prove to be beneficial.

In terms of general motivations, I believe that working in a team will be very helpful for my future as a computer scientist. I look forward to bettering my communication skills, learning how to hold both myself and others accountable in a team-focused way, and being able to communicate with others to solve problems.

2.3.2.2 Ideas

This project has two main components: the GNNs and the cyber-physical systems (networks) whose anomalies we are detecting. I believe that to succeed, we need to ensure that everyone has a clear understanding of what the project is and how GNNs could solve it. Since

none of the group members have previous experience with GNNs, that is the first thing that will be tackled. As I mentioned in the motivation session, a large part of this project will deal with performing literature reviews. These will be extremely important to familiarize ourselves with concepts that we do not understand (the first being GNNs), as well as go deeper into our knowledge of concepts we do understand. Our sponsor has provided us with many resources to go about this effectively.

Once we have become familiar with both the structure of the networks and GNNs, we should begin to look into what defines an anomaly. There are various types of anomalies in a graph, such as node anomalies and edge anomalies. We can define an MVP from one of those and move towards testing other anomalies.

The first step in most, if not all, machine learning projects is preparing the data. Our sponsor has provided us with various Pcap files that we will use to identify potential input features to associate with our graph. The length of the packet or the type of protocol used between connections are both examples of potential input features.

Part of our goal for this project is to produce something that hasn't been done yet and outperform current state-of-the-art GNN architectures. An example of this that the team has discussed is how we should take into account loops. Current architectures explore a depth that does not exceed the graph diameter to not look at data that we already have, but that may miss a node that is connected to itself. My role on previous teams has varied depending on what type of project I am working on, but I would say that I possess a good ability to creatively identify edge cases such as the one above, and I look forward to solving those within this project.

2.3.3 Landon Russell

2.3.3.1 Motivation

A lot of my motivation for our project stems from the chance to learn and explore a relatively different field of programming experience than I am used to. In my current internship and past school experience, I have been exposed to many different topics in computer science, but have never gone this deep into cybersecurity topics that have a large amount of mathematical components to go along with it. When working with graphs in the programming contest class I took last year, as well as practicing for technical questions like leetcode, I have always wondered how they could be applied practically in the real world. I am excited to see how they work in the sense of graph neural networks and how they work to detect anomalies in data. So far, from initial research as well as discussing the details of the project with my team and our sponsor, it has made me even more excited to get started on our task laid out by the sponsor.

The cybersecurity field is something I have wanted to explore more, and I feel like a task involving understanding cutting-edge systems and attempting to produce something more efficient in regards to something as complex as graph neural networks is a great way to put my foot forward. Even if I don't continue on to pursue a position in cyber security for my career, it is still always a great opportunity to expose myself to as many options as possible to find what I enjoy the most within the field

In a practical sense, this is also a great opportunity to possibly pursue a publication as well as have a great project to add to my resume. I personally want my resume to include advanced projects that I am passionate about. Having a passion that I see myself having in our GNN can

give me plenty to talk about when applying to certain positions that will find something like what we are doing with it interesting. Having a publication in something like this is one thing that I have always wanted to attempt, and this could be a chance for that as well. Both having a publication and having another strong project for my resume will play a role in boosting my professional confidence and overall confidence in the realm of my Computer Science career.

2.3.3.2 Ideas

Since the concept of our project is one that has been explored in the past, and the concept of graph neural networks being used for anomaly detection isn't new, it is very important that we focus on getting a broad view of similar material that has already been completed. This means that extensive research and planning are necessary in order to reach the goal that the sponsor expects of us. To start, the documents sent by our sponsor as well as other sources we find while doing our own research, will greatly enhance our ability to understand the content we are working with.

Reading extensively through these sources will allow us to know what tools we have at our disposal that can assist us in implementing, as well as understanding which of these tools are open to expanding on in order to make our completed Graph Neural Network unique and better than the cutting edge technology that is already being implemented by different companies to this day. Basing our initial investigation on a paper that our sponsor said was a good starting point is the best idea. The paper gives an overall view of the current status and challenges of graph neural networks that are implemented today.

Following research and having a strong foundation and knowledge of graph neural networks, it is important to have a set plan and guidelines that we will follow to complete our goal. We need to know what is going to be coded and have a physical implementation, as well as what ideas we will continue to look over and explore throughout the semester. All of this will be a part of what goes into our paper to complete by the end of the semester.

From the research I have gathered, my high-level idea of how we could start is to first gather and prepare our data. Thankfully, our sponsor has provided us with data sets that can be implemented. Next, we need to create the overall architecture of our GNN. This can be accomplished by having multiple Graph Convolutional Layers, which our sponsor briefly explained to us. With this, we can decide the number of layers as well as what activation function we can use. After, we have to train the Graph Neural Network we have created. With this, we will then use the GNN model to make predictions and update the model's weight based on loss. Finally, we will evaluate the model.

2.3.4 Mukundh Vasudevan

2.3.4.1 Motivations

My motivation for this project is to learn more about graph neural networks. As I have not learned a lot about neural networks in general, this project felt like a good opportunity to learn about graph neural networks. Another thing that intrigued me about this project was the cybersecurity aspect, as I have experience in networking and cybersecurity.

This project originally piqued my interest because, at the time, I was working on getting a Suricata intrusion detection system on my home network. This intrusion detection system is a traditional system that uses conventional computing to find anomalies in the network. When I saw there was a project about finding those anomalies with graph neural networks it was a great fit.

My interest in the project drastically increased after talking to our sponsor about graph neural networks. He put a lot of emphasis on creating a state-of-the-art graph neural network and the reasoning for why we should. This will be a great improvement to cybersecurity as well as researching graph neural networks.

I have lots of experience in networking, so I believe I have lots of experience in reading the network topology and understanding it thoroughly. I have also spent a lot of time understanding packet capture (PCAP) files, so I think I can bring a lot to the table. Being able to understand this raw network data makes it so that I can understand.

I know that this project is essentially creating state-of-the-art network security. As I would like to get into a cybersecurity career, this seemed the best project to do. The GitHub that was given showed a simulation network that is what we are going to be using. After looking at this repo, I liked the network that it had, so I wanted to join this project.

2.3.4.2 Ideas

Finding anomalies in a graph neural network has already been created. Because of this fact, we have to be able to differentiate our project, making

ours better than what is currently state of the art. To know how anything works and what currently exists, a literature review needs to be conducted to understand what is going on.

First, I need to get knowledge of graph neural networks and to do that, we have gotten resources from our sponsor to be able to learn this. There are many frameworks for graph neural networks, but we will be using pytorch and pytorch graph neural network frameworks.

To aid with the fact that we need to do a literature review, our project sponsor gave us links to papers which include papers from papers with code, Google Scholar, and a paper about the state of anomaly detection using neural networks. This told us about what ways these neural networks were created and the many ways we could make them.

Another part of the project is to understand the networking, the packet capture files, and the security based on this. The network topology that we will be using is the simulation created by Pacific Northwest Laboratories. We will need to understand the full network to make the ideal graph to be able to create good data for the neural network.

2.3.5 Nicholas Lonnon

2.3.5.1 Motivations

My main motivation for this project is to get a passing grade. I would also like to get a better understanding of how neural networks work. There is a lot of math that comes into optimizing neural networks and a lot of research that is needed to do that math, and this project seems like it will

rely heavily on that research. I hope to develop a good enough understanding of how neural networks work to the point where I can get a job. I also hope that this project will look good on my resume.

I am interested in neural networks because they seem pretty neat. I could give chatGPT health-related symptoms, and it would probably diagnose the cause and a solution faster and better than a doctor, and I don't need to go anywhere.

I've mainly messed around with natural language processors, but I would like to expand my knowledge on the subject. I've never worked with Graph Neural Networks before, but I am interested in learning more about them and implementing them. I've made my own multi-attention models, but I have never had a GPU to make something interesting.

2.3.5.2 Ideas

We are taking Propagation Code Analysis Program data and using that data to detect anomalies using Graph Neural Networks. There are different architectures of GNNs, and depending on the architecture of our Graph Neural Networks will affect how we format our PCAP data as a graph.

One Graph Neural Network type architecture we can use is a Graph Convolutional Network. It uses a multilayer perceptron like a Convolutional Neural Network, meaning that it has layers of hidden states that it uses to generate an output from input. But its hidden states on each layer correspond to a node on the network, and each node is represented by some value. Each hidden state is equal to the ReLU of the summation of each adjacent node in a function. This function is defined as the product of the

corresponding hidden state in the last layer multiplied by a weight that is changed by backpropagation over the root of the product of the connected nodes. The issue with this Graph Neural Network is that it doesn’t take into account the properties of the edges other than the nodes that it connects.

A Graph Neural Network is similar to Graph Convolutional Networks in how it connects the different layers of the hidden states between their corresponding adjacent nodes, Message-passing neural networks are able to take into account edge properties. It has a hidden state in each layer that corresponds to each of the nodes like in a Graph Convolutional Network, but it accumulates the sum of the adjacent nodes by passing the previous hidden layer of the current node and adjacent node along with the edge properties between the corresponding node into a messenger function. It then takes the sum returned from that messenger function and passes it into the vertex update function along with the corresponding hidden state on the previous layer. The messenger function acts as an accumulator, and the vertex update function acts like an activation function

Like a Convolutional Neural Network input is fed into these hidden states to generate a predicted value, and if the model is training, this predicted value is compared to the actual value and backpropagation through the weights of the model to generate a value closer to the actual value.

Another more modern Graph Neural Network is a Graph Transform Network. This version, unlike the other two, is based on the principles of the Transformer Model, which is typically used for Natural Language Processing. instead of words, it takes in nodes and, optionally, edges of graphs. Unlike Transform Models in Natural Language Processing, the Graph Transform

Network in each of the attention heads compares each of the adjacent nodes with the node that it is looking at and pools them into a sum.

2.3.6 Santiago Rodriguez

2.3.6.1 Motivations

I am motivated to learn more about graph neural networks and how they combine both machine learning tools with graph theory. Beyond my intellectual curiosity to know how graph neural networks operate, I am also interested in seeing how these technologies can be applied to the real world. Since this project will be looking at applications of machine learning in the cybersecurity space, I believe this project will be a great experience to explore the kind of network and security problems being posed and see how useful machine learning and graph theory are at solving these problems.

This project, which aims to study neural networks and graph theory in the context of network security, would also parallel my other research project in computer science. In particular, over the summer, I worked on a project studying the transformations that go between continuous spaces (like the real line) and programs (like lambda calculi). I aimed to uncover what it would mean to say that a function from reals to programs is continuous, and moreover, what it would mean to say that the function is differentiable.

This would have applications to the study of neural networks since one can think of the training algorithms as transforming programs (in this case, neural networks) into numbers (in this case, the error cost) and then minimizing the cost. Knowing what these transformations look like and the

kind of properties they have would prove very useful in giving a rigorous analysis of how neural networks work. Using tools from domain theory, representation theory, and topology, I was able to present suitable definitions for continuous transformations and a class of continuously differentiable transformations that do not require imposing a linear structure on the space of programs.

In either case, both research projects require a study of neural networks and how they function in order to improve their functionality for specific tasks. For this senior design project, being able to come up with an algorithm that can reliably detect network anomalies is part of the reason I joined this project. The other part, given that this is a research project, is to hopefully end with a peer-reviewed publication so others can see what we've done.

2.3.6.2 Ideas

Since this is a research project, first and foremost, doing a literature review is essential to get an idea as to how the problem has been framed and dealt with in the literature, what tools are available, and what are the pros and cons of each approach. Even so, I already have some idea as to how to approach the problem of identifying network anomalies.

Since the goal is to detect network anomalies using neural networks, we first need to define and collect the data that will be used to train our network. Moreover, we need a way to standardize this data so that arbitrary-sized communication networks can all be analyzed using a single algorithm instead of having to retrain the model for each possible network

architecture. And lastly, we need a method of training the neural network so that it reliably converges to a suitably optimal solution.

In regards to collecting the data, our sponsor has already provided us with resources to synthesize network attacks via an open-source simulation project. In addition, there are standard databases for cybersecurity researchers to work on, which we will be utilizing.

In regards to standardizing the data, the graph structure itself need not be explicit in the data being passed. By the nature of how graph neural networks work, the number of update iterations corresponds to how large of a vertex neighborhood one considers when making predictions. Thus, the only data standardization that needs to be done will be the metadata associated with each node. We could take the full packet data recorded during network traffic and use that as the metadata for each node. Since time is an important factor in identifying network anomalies, one possible solution would be to only consider a fixed interval of time where we determine duration by trying multiple intervals and comparing accuracy and precision in identifying network anomalies.

Finally, in regards to training, we could use the standard gradient descent algorithm with some optimizer like Adam. To calculate the gradient, it may be easier to use an auto differentiation package than to calculate the gradient by hand. Thus, this will lighten the workload if we ever choose to change the underlying update function.

2.4 Societal Impacts

2.4.1 National Security

As discussed in the previous sections, there are serious humanitarian consequences in the case of a highly successful cyberattack on the United States smart power grid. In a worst-case scenario, a cyberattack on the power grid could be considered an act of war. Actors like Russia, China, and North Korea could easily sabotage an insecure power grid if our countries ever engaged in a conflict or war. It is easy to imagine such an attack damaging the defense systems and infrastructure in our nation. Some experts have proposed that many of these countries are already capable of inflicting debilitating damage to the United States power grid but only refrain from doing so for political and peacekeeping reasons.

However, this issue does not only impact American national security. One notable attack outlined in the report from the Idaho National Laboratory mentions the December 2015 attack on the Ukrainian power grid, attributed to a Russian military hacker group. [5] Since then, cyberattacks on the Ukrainian power grid have only increased with the progression of the Russo-Ukrainian war and continue to this day.

As modern countries transition to the smart grid system, they are opening themselves up to increased sabotage of their infrastructure from other nations. For this reason, it is important to protect our power grid and other related systems from a cyberattack scenario in a national defense context.

2.4.2 Other Actors

Besides other nations threatening the US power grid, it is worth mentioning some other bad actors that may launch similar cyberattacks. This includes terrorists, hacktivist groups, cybercriminals, and other fringe groups. It is important to consider these organizations since they add to the chaotic and dynamic landscape of the cybersecurity problem scope, where technique, technology, and motivation can quickly change.

2.4.3 Public Safety and Economy

In cities that are not prepared, the most vulnerable people are those who depend on energy to power hospitals or health-related technologies for day-to-day use, including communication between first responders. Power outages would negatively impact digital communication and the economy, especially in an industrialized country with many white-collar workers who rely on technology and power in order to complete work at their jobs.

Furthermore, the implications of a cyberattack extend beyond mere disruption, as the type of outage, targeted system, and nature of the attack could potentially result in lasting physical damage to the affected infrastructure, particularly in the long term. This broader perspective underscores the multifaceted and interconnected nature of cybersecurity challenges.

Considering the integral role that power systems play in supporting various critical infrastructures, such as water and sewage systems, the repercussions of a cyberattack on the power grid can have cascading effects, impacting essential services that rely on a stable and secure power supply.

Recognizing these interdependencies emphasizes the importance of comprehensive cybersecurity measures to safeguard not only the targeted systems but also the broader network of interconnected infrastructures vital to the functioning of the United States.

A lack of access to clean drinking water or operational sewage systems could become a massive issue in a long-term power grid failure. Any combination of these socio-economic issues could ultimately lead to social unrest in a long-term power grid outage. It is safe to say that such a scenario would be catastrophic.

The significance of cybersecurity software capable of real-time attack detection cannot be overstated, particularly in bolstering governments' capacity to shield their citizens from the ramifications of power grid outages. The potential advantages, especially for the U.S. government in the realm of national security, are substantial. The ability to promptly identify and respond to cyberattacks as they unfold is a critical aspect of fortifying the power grid systems.

Consequently, our research endeavors stand to help advance current technology for defending power grids against malicious cyber threats. By contributing to the development of more robust and responsive cybersecurity solutions, our work aligns with the broader objective of enhancing the security posture of critical infrastructures, thereby safeguarding essential services in society.

With increasing global reliance on networks and cyberphysical grids, ensuring the security. This research will not only help efforts in cybersecurity but will also contribute significantly to national security. Not only are we

looking to help immediate threats to also lay down a strong foundation for future infrastructure.

3. Project Characterization

Our group has been tasked with creating a novel Graph Neural Network architecture or enhancement for state-of-the-art network anomaly detection. This project is sponsored by Branden Stone, and we are working in collaboration with the Georgia Tech Research Institute (GTRI). We are studying the network data transmitted between cyber-physical systems used for infrastructure in the United States. The reasoning behind this is that networks lend themselves nicely to be represented in a graph data structure, with the components of the network being nodes and the data transmitted being edges.

GNNs take a graph as their input and consider the graph structure by looking at a specific node's neighbors and the relationship between them. With this architecture being the end goal of our project, our initial focus was on creating simple models such as random forest and basic neural network classifiers on packet data in order to become more familiar with our data. This was supplemented with constant research on GNNs to make a connection between our current and future works and prepare ourselves for GNN implementation.

Our ideal final deliverable would be able to perform anomaly detection using a simulation environment called Network Attack Testbed In [Power] Grid (NATI[P]G) developed by Pacific Northwest National Laboratory (PNNL) [2]. However, the PNNL is continuously developing this simulation environment, and a stable release of this software has yet to be developed. This environment produces simulated packet data and follows the same structure as a packet capture (PCAP) file. For our initial proof of concept

classifiers, the UNSW-NB15 dataset [1] provided by UNSW Sydney was used.

This dataset contains raw packet data and attack types gathered by a Network Intrusion Detection System (NIDS). The main difference between our original classifiers and the graph neural network lies in the fact that with our original classifier, our focus is on individual packets, whereas our final product will observe any subset of the packet data. This allows us to look at the connection between packets which opens a new door to types of attacks that can occur. This would be represented as the edge of a graph, and we would use that to identify an anomaly.

Within GNNs used for anomaly detection, we eventually decided to use the GraphSAGE architecture [35] as detailed in section 4.1.13.2 of this document.

3.1 Objectives and Goals

In order for our team to successfully build up to a novel graph neural network architecture for network anomaly detection, we divided our objective into two major milestones. The first milestone focuses on packet-level anomaly detection, that is, using only a single PCAP packet to predict anomalous behavior. The second milestone focuses on network-level anomaly detection, that is, using a suitably large time frame of PCAP data to predict anomalous behavior.

For the first milestone, we developed a two-stage pipeline for packet-level anomaly detection by the end of Senior Design 1 (December 3). Specifically, the pipeline consists of (1) a decision tree classifier and (2) a

neural network classifier. The decision tree classifier looked at an individual PCAP packet and categorized it as either normal or anomalous. Packets that were labeled as anomalous are then fed into the neural network classifier. The neural network classifier looks at an anomalously-labeled PCAP packet and categorize it as one of the specific types of attacks. In particular, this includes: distributed denial of service attacks, injection attacks, and parameter attacks.

This first milestone is intended to teach us exactly what features in the PCAP data are most relevant in anomaly detection. This also serveed as a minimal working demo showcasing the utility of artificial intelligence models in predicting network anomalies. The data we collect from this first milestone then serves to inspire the design choice of the second milestone. The code from this milestone did not make it into our final codebase as we began to focus our effort on expressing work with the graph neural network and obtaining a running simulation environment.

For the second milestone, we developed a graph neural network for network-level anomaly detection by the middle of Senior Design 2. Based on the relevant features identified in the first milestone via analysis and domain-specific knowledge, we trained a graph neural network to autoencode the communication network into embeddings that place normal nodes within a specific threshold and anomalous nodes outside that threshold. The hope is that focusing on the features identified in the first milestone helped optimize the graph neural network in considering packet-level anomaly discriminators while finding a suitable embedding to encode the network activity.

As a stretch goal, we aimed to complete the project with a graph neural network architecture that improves on the state-of-the-art in network anomaly detection. Provided that our model works and there is sufficient novelty/interest as determined by our sponsor, our team will author a paper intended for publication. However, we did not reach this stretch goal in the given time and this is an area where future teams may be able to contribute.

3.2 Specifications and Requirements

As we are working on a relatively large research and computer science project, working with Jira in an Agile format has become a necessity. Thankfully a couple of our team members are already familiar with the Jira platform, and they were able to take on the responsibility of setting up the Jira board and getting the rest of the group familiar with the platform and concepts of sprints. This task seemed trivial at first, given that our project scope did not have a very specific design, but it ended up making the task setup much more complicated.

All tasks have been created as user stories, and they are assigned to their designated epics, which were the following: Random Forest on UNSW, Supervised Neural Network on UNSW, Graph Neural Network on UNSW, Sim Environment Setup, and Research. Given that this is not a traditional software project, user story delegation has been difficult since there is significant overlap between tasks.

All the team members are worked on getting familiar with the technology by completing similar small machine-learning projects and doing individual research. Learning how to use Jira effectively for Agile project management and laying out a strong foundation for how our project planning should look moving forward was essential for a successful project.

Overall, the requirements for this project are (1) a database of instances of network activity attacks, (2) a PCAP file parser for extracting the packet data from these network activities, and (3) a graph neural network that takes in the packet data and network structure and can practically detect anomalous behavior. As this is a research project, the final (weak) requirement is that the graph neural network should have made noticeable improvements from state-of-the-art network anomaly detection models.

In addition to these requirements, the specifications include: (1) sourcing from the UNSW dataset and NATI[P]G simulator [2] to train our models; (2) a Python script for parsing; and (3) a PyTorch implementation. Additional cloud computing for network training would also greatly improve the project as it would speed up training and provide a reliable tool for team members to work on their individual parts. Our team was able to train on a standard laptop PC. However, additional computing may be necessary for future development.

3.3. Concept of Operations

Our project applied the GraphSAGE [35] graph neural network to a simulated cyberattack scenario from a power grid to detect anomalies within the system that could be potential attacks. As stated in the paper Graph Anomaly Detection With Graph Neural Networks: Current Status and Challenges: “Graph anomalies are patterns in a graph that do not conform to normal patterns expected of the attributes and/or structures of the graph.” [1]

Because of the nature of the graph structure, a graph neural network is an ideal machine learning representation to use for a cybersecurity scenario, with the graph anomalies becoming the attacks on the network that we aim to detect. The graph structure itself is leveraged in order to solve this problem. As our sponsor and research mentor specified, our neural network models must be compatible with PyTorch, and we must test our neural networks on simulated data from the Network Attack Testbed In [Power] Grid (NATI[P]G) simulation. [2] Our sponsor has provided us with the open-source simulation code, which is required to run in order to obtain this data. [1]

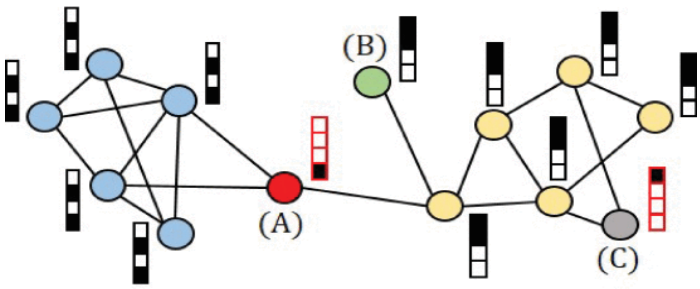


Figure 1. Example of a graph neural network. [1]

The user is able to input data into our graph neural networks, and it predicts anomalies based on this data. The application pipeline takes in packet capture (pcap) files, analyzes the network, and detects anomalies using graph neural networks. As a stretch goal, our team also planned an integration of the pipeline code where the user will not interact directly with the application, but they will get an alert when an anomaly is detected. This is a potential goal for future teams. The anomalies that we detect are

outlined in the UNSW dataset and include distributed denial of service, fuzzers, analysis, backdoors, exploits, generic, recon, shellcode, and worms.

With Cyber-physical infrastructure becoming more common in the United States, the security of these systems is an increasingly important concern. The commercial use for this is to improve Intrusion Detection Systems for distribution power grid networks using state-of-the-art simulators.

We used Python, including but not limited to the PyTorch package. Our sponsor has recommended the use of other packages, such as Sci-Kit Learn to begin to obtain a preliminary understanding of the problem we are trying to solve. We also ran our cyberattack data simulation, Network Attack Testbed In [Power] Grid (NATI[P]G), which is written in C++ and Python and provided by The Pacific Northwest National Laboratory on GitHub. (NATI[P]G) is a "co-simulation environment for distribution power grid network using state-of-the-art simulators." [2]

The repository can be located at https://github.com/pnnl/NATIG. The simulation uses the language C++, and it uses the following libraries: XInclude, cmake, helics, ns3, PParse, MemParse, PSVIWritter, Redirect, SAX2 Count, y-example-debug, SAX2Print, Xample-debug, XInclude, and idle. [1] As previously mentioned, the team was able to obtain a working simulation environment setup, but was unable to test and train on this data due to a lack of a stable release from the team at PNNL.

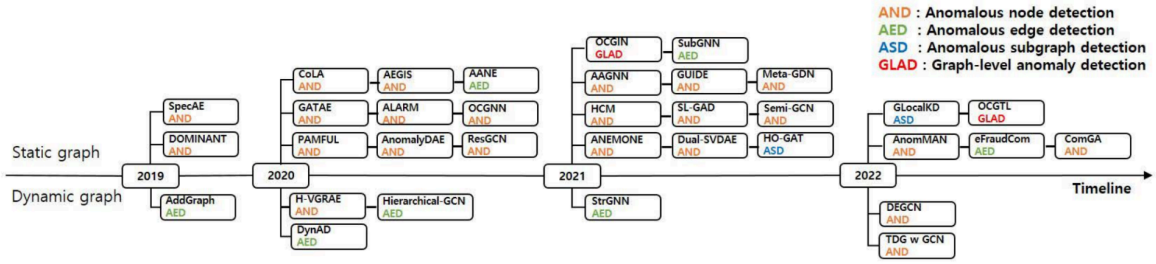


Figure 2. Timeline representing previous anomaly detection Graph Neural Network models. [1]

Other technologies we may used include Docker and Pip Package Manager. We also used Box, which is a file-sharing platform that we will use to store important documents throughout our project, including papers we read and plan to incorporate into our own work. This technology was selected by our research mentor, who also has access to our shared Box space.

4. Design Documentation

4.1 Research and Investigation

Based on our first meeting with our sponsor and research mentor, we recognized that our goal is to find the best and most cutting-edge graph neural network. The group has successfully implemented the GraphSAGE network. We hope that work of future teams incluse creating a new type of Graph Neural Network that extends this network and implementation on the Network Attack Testbed In [Power] Grid (NATI[P]G) simulation. [2] Our sponsor recommended multiple resources, including Papers With Code and Google Scholar. We have also used arXiv in order to make the most of our literature review process. Our sponsor has also suggested one paper as a starting point that gives a bit of an overview of the field of graph anomaly detection using graph neural networks as it stands today. [1] Looking at the papers that cite this and the papers that it cites was be a great starting point to read up on this field.

Since this project is heavily centered around understanding what a graph neural network is, how they are used, and what the cutting-edge technology around them is, a lot of the early stages of this project revolve around researching. Although it might not seem like physical contributions, our sponsor agreed that this extensive research and comprehension is absolutely necessary when moving into the implementation phase during the second semester. Initially, our group wanted to understand three major topics that contribute to implementing our Graph Neural Network in a practical form, Graphs, Neural Networks as a whole, and Pytorch/ML Implementation. Our sponsor provided us with documentation and resources for all, starting with Neural Networks and how they will be used in our

project. Providing how each topic will apply to the overall project is just as important as understanding the fundamentals.

4.1.1 Networks

With the main goal of the problem being to detect anomalies within network intrusion systems, becoming familiar with a network and its components is an essential part of the process. This helped us understand our dataset and the features we planned to use during classification. A packet is a data structure containing a header, payload, and trailer. The header is composed of metadata related to the data in the packet. The payload is the actual data within the packet. Our data specifically does not contain a trailer because it is an IP packet.

4.1.1.2 Packet Payload

Packets can be used to create a network by using the devices that they outline and the data transmitted between them. The most important information contained within a packet as it relates to the UNSW-NB15 data is outlined below. [3]

IP Address	Numerical address assigned to any device connected to the internet.
Port	Number associated with protocol that sends or receives data within a device.
Transport Layer Protocol	Set of rules for devices to communicate. Ex: UDP, TCP
State	Current state of network. Ex: ACCEPTED, CONNECTED, CLOSED
Transmission Control Protocol (TCP) Window	A rule that helps the device know how much data it can handle.
Time To Live (TTL)	The amount of time that a packet exists on a computer before it is discarded in terms of how many routers it passes through.
Load (bytes per second)	The amount of data being transferred.

Packet Loss	How many packets failed to get to the destination. (typically represented as a percentage of total packets)
Average Packet Inter-Arrival Time	Metric that measures the average time between packets arriving in succession.
Jitter	A variation in the amount of time that it takes for packets to reach a destination.
Application Layer	Layer that lays out communication protocols. Top-most layer or Open System Interconnection (OSI) model.

4.1.2 UNSW-NB15 Dataset

To best go about detecting anomalies within network data, we usedthe most widely used dataset in the field of network intrusion by the instruction of our project sponsor. The UNSW-NB15 dataset, gathered by NIDS at UNSW in Australia provides extremely detailed network traffic data. [3] Not only does it include normal network traffic but also various network attack types, making it a very valuable dataset for machine learning used in network security. The dataset contains 49 features and 9 different attack types.

The nine types of attacks are the following: Fuzzers, Analysis, Backdoors, DoS, Exploits, Generic, Reconnaissance, Shellcode and Worms. Fuzzers are scripts created by attackers sending large amounts of data to a network. The goal of this is to find vulnerabilities within an application and use those security flaws in a larger attack. Analysis attacks look at network data to learn more about the environment that they are targeting. This newly gathered information can help identify any weak points within the application.

Backdoors are created by attackers from within the application to allow for later entrance. This means that even after an initial patch that may not allow for the first breach, attackers can find their way into the application. Denial of Service (DoS) attacks work by providing the application with excessive traffic such that it does not have enough resources to withstand it. This can crash the system and allow the attacker to understand the vulnerabilities within that system. Exploits use known weaknesses in the application to attack it.

Generic attacks do not fall into a specific category or follow a pattern. By utilizing techniques that are not common or widely used, generic attacks simply aim to find a vulnerability within the network. Like Analysis attacks, Reconnaissance attacks deal with network data, although their main purpose is to gather it rather than draw conclusions about it. It is as if Reconnaissance attacks lead to analysis. Shellcode is a type of injection attack that looks to gain access to the application's shell. Access to the shell has value as it allows the attacker to run potentially dangerous commands and gain data while unauthorized. Worms are pieces of malware that can self-replicate to attack as many systems in as little time as possible.

Monitoring network traffic to better our understanding of what packet data can lead to these types of attacks helped us identify anomalies within networks. In practice we used simulated data with a similar structure to the UNSW-NB15 dataset. Our first step was to use a Random Forest and then a generalized basic neural network to determine what the most valuable features are.

4.1.2.1 Features

There were 49 features, including the label, outlined below: [3]

No.	Name	Type	Description
1	srcip	nominal	Source IP address
2	sport	integer	Source port number
3	dstip	nominal	Destination IP address
4	dsport	integer	Destination port number
5	proto	nominal	Transaction protocol
6	state	nominal	Indicates to the state and its dependent protocol, e.g. ACC, CLO,

			CON, ECO, ECR, FIN, INT, MAS, PAR, REQ, RST, TST, TXD, URH, URN, and (-) (if not used state)
7	dur	Float	Record total duration
8	sbytes	Integer	Source to destination transaction bytes
9	dbytes	Integer	Destination to source transaction bytes
10	sttl	Integer	Source to destination time to live value
11	dttl	Integer	Destination to source time to live value
12	sloss	Integer	Source packets retransmitted or dropped
13	dloss	Integer	Destination packets retransmitted or dropped

14	service	nominal	http, ftp, smtp, ssh, dns, ftp-data ,irc and (-) if not much used service
15	Sload	Float	Source bits per second
16	Dload	Float	Destination bits per second
17	Spk Ts	integer	Source to destination packet count
18	Dpkts	integer	Destination to source packet count
19	swin	integer	Source TCP window advertisement value
20	dwin	integer	Destination TCP window advertisement value
21	stcpb	integer	Source TCP base sequence number
22	dtcpb	integer	Destination TCP base sequence number

23	smeansz	integer	Mean of the ?ow packet size transmitted by the src
24	dmeansz	integer	Mean of the ?ow packet size transmitted by the dst
25	trans_depth	integer	Represents the pipelined depth into the connection of http request/response transaction
26	res_bdy_len	integer	Actual uncompressed content size of the data transferred from the server's http service.
27	Sjit	Float	Source jitter (mSec)
28	Djit	Float	Destination jitter (mSec)
29	Stime	Timestamp	record start time
30	Ltime	Timestamp	record last time
31	Sintpkt	Float	Source inter packet arrival time (mSec)

32	Dintpkt	Float	Destination interpacket arrival time (mSec)
33	tcprtt	Float	TCP connection setup round-trip time, the sum of ’synack’ and ’ackdat’.
34	synack	Float	TCP connection setup time, the time between the SYN and the SYN_ACK packets.
35	ackdat	Float	TCP connection setup time, the time between the SYN_ACK and the ACK packets.
36	is_sm_ips_ports	Binary	If source (1) and destination (3)IP addresses equal and port numbers (2)(4) equal then, this variable takes value 1 else 0
37	ct_state_ttl	Integer	No. for each state (6) according to specific range of values for source/destination time to live (10) (11).

38	ct_flw_http_mthd	Integer	No. of flows that has methods such as Get and Post in http service.
39	is_ftp_login	Binary	If the ftp session is accessed by user and password then 1 else 0.
40	ct_ftp_cmd	integer	No of flows that has a command in ftp session.
41	ct_srv_src	integer	No. of connections that contain the same service (14) and source address (1) in 100 connections according to the last time (26).
42	ct_srv_dst	integer	No. of connections that contain the same service (14) and destination address (3) in 100 connections according to the last time (26).
43	ct_dst_ltm	integer	No. of connections of the same destination address (3) in 100 connections according to the last time (26).

44	ct_src_ ltm	integer	No. of connections of the same source address (1) in 100 connections according to the last time (26).
45	ct_src_dport_ltm	integer	No of connections of the same source address (1) and the destination port (4) in 100 connections according to the last time (26).
46	ct_dst_sport_ltm	integer	No of connections of the same destination address (3) and the source port (2) in 100 connections according to the last time (26).
47	ct_dst_src_ltm	integer	No of connections of the same source (1) and the destination (3) address in in 100 connections according to the last time (26).
48	attack_cat	nominal	The name of each attack category. In this data set , nine categories e.g. Fuzzers, Analysis, Backdoors, DoS Exploits, Generic,

			Reconnaissance, Shellcode and Worms
49	Label	binary	0 for normal and 1 for attack records

We were given two datasets to be able to test our neural network. The first was a dataset curated by UNSW, which is a giant set of PCAP files. The second dataset is the simulation environment of a power grid network setup created by the Pacific Northwest National Laboratory. [3] To be able to run code in the simulation environment, we have tested running the Docker container and have been able to interact with the packets used by PyShark.

Our sponsors' stretch goal for this project is for us to improve graph neural networks. With the research that we have done, we believe that future teams can build on our work to develop a novel graph neural network that elaborates on our existing GraphSAGE implementation.

The differences between the two datasets are useful to simulate our test environment. The UNSW dataset has 2540044 packets, which have nine different types of attacks. This is very useful as this is simulated to look like data an IDS is supposed to detect.

The Pacific Northwest National Laboratory simulation is good to test as this is an environment that is live and acts as the network of a power grid operator. This is something that we do not have access to, so this is a good simulation of how it might look live.

4.1.3 Simulation Environment

The Pacific Northwest National Laboratory simulation environment named Network Attack Testbed In [Power] Grid was set up to run on docker, letting us sniff packets in this network. This simulation environment can give three types of attacks. These are injections, parameter changes, and distributed denial of service attacks. We created a Python script that sniffs the packets using PyShark and turns them into CSVs the same way that we interfaced with the UNSW data. We did this so that it would be easy to migrate the code over to this. This simulation environment is a great place to test the efficiency and reliability of the graph neural network, as we can change the parameters of the attack.

The final goal for this project is to get our graph neural network running on this simulation environment and to get all three of these attacks accurately detected all the time. False positives are better than false negatives, as we would rather detect all the problems and flag a few benign packets as malicious than let a malicious packet through.

4.1.4 Neural Networks

Our sponsor provided us with detailed articles on Neural Networks and they each gave a high-level view and example of how NN’s are created and what they do when implemented. Having examined three articles detailing the foundations of these systems, it's clear that we must first unpack the basics before we try to move into more sophisticated applications regarding

our overall project involving Graph Neural Networks and how they can detect anomalies within data.

Looking at these articles centered around providing information on NNs as a whole while also focusing on being able to understand the information that could apply to the general goal of implementing a GNN allowed for an easier time in the research process by getting a solid foundation of all necessary details.

Neural networks are designed based on how the human brain works. They have many connected parts called neuron-like units. These units are tied together by weights. The weights are what decide how strong the influence of a unit is on another unit. When you change the weights you can find the perfect one to make the prediction capabilities of the network better. Gradient descent and backpropagation are tactics to use in order to make the correct changes to weights.

Neural Networks are clearly going to be a huge part of our overall project. The applications come with how well we can understand the foundation of GNNs which is an NN. By understanding how to work with them effectively, we can create a graph neural network that is able to spot anomalies in data with high accuracy and speed.

Activation functions are used by the neurons of a neural network to make decisions on how they respond to the data passed through them. When talking about the most popular activation functions, both Sigmoid and ReLU are popular choices. Sigmoid reduces real numbers to a value between 0 and 1, while ReLU is a nonlinear model that helps with an issue called vanishing gradient, which is when a gradient used to update the network becomes so small that it “disappears.” They play a role in helping the

network figure out and learn intricate data patterns. However, each activation function has its different usage scenarios and won’t always be the right pick for every situation.

Selecting the correct activation function for our project is a very important task when we implement our Graph Neural Network in code. It can make the difference between a network that spots anomalies correctly and efficiently and one that struggles with even simple tasks.

Just like how we sometimes learn things but don't always apply them correctly, neural networks can also have these unfortunate issues. Even if a network performs really well on one task, like having a 98% accuracy rate when recognizing images, it can still make obvious mistakes that cause difficulty. For instance, while we'd expect it to pick up on clear patterns, it might sometimes get distracted by random ones that don't really matter. It's great at noticing something within data but has tendencies to identify false positives.

This is an important lesson for our project. When building our graph neural network, we need to focus on training well enough to detect real anomalies and not target irrelevant data. Although neural networks can be very powerful for many different tasks the research in these articles and elsewhere shows that they are not perfect. Older types of networks had their own set of challenges, like being more rigid in their learning. They couldn't easily transfer what they learned from one part of the data to another, which provided some difficulties. Thankfully, newer versions like convolutional neural networks have done the job of being multifaceted enough to get through these difficulties and have addressed many of these limitations.

Understanding past limitations of Neural Networks can help us avoid these challenges in the product that we create. It also allows us to identify what areas we can look at expanding on when trying to beat the cutting-edge technology that's implemented today. Our objective is to build a graph neural network that either has something that others don't or goes above and beyond on one piece of functionality.

There are a ton of tangible, real-world applications of neural networks, a lot of the time, we just don’t typically notice them. Tech like image recognition, similar to the example we saw in the paper provided by our sponsor, as well as cybersecurity, which is usually done behind the scenes. The ability of neural networks to process vast amounts of data and draw meaningful conclusions is extremely helpful in fields like cybersecurity or, more specifically, anomaly detection.

By integrating our anomaly detection mechanisms into real-world systems, we offer practical solutions. Whether it's spotting irregular patterns in financial transactions or identifying unusual behavior in security systems, our graph neural network could be an invaluable asset.

Neural networks are taught through continuous learning. They first set a foundation through initial training but need to also be exposed to varied data over time. This ensures that they adapt, evolve, and remain relevant when looking at different types of data sources or are used in different scenarios.

As we move forward with our graph neural network, ensuring it has opportunities to learn from new data keeps it sharp in its anomaly detection tasks. Our network continuously improves while being exposed to more data.

Neural networks, with their neuron-like structures and learning capabilities, are at the forefront of modern technology applications. By understanding their basics, strengths, and limitations, we're better equipped to harness their potential in anomaly detection. When we took the next steps in creating a graph neural network, we used these lessons as our guideposts.

To grasp GNNs, we first establish an understanding of a basic neural network architecture. Neural networks are based on the structure of the human brain and have components known as neurons to make predictions. The foundation of a neural network can be built upon to create many specialized neural networks, such as Convolutional and Recurrent neural networks.

Neurons in a basic neural network are composed of some input signal. The input of a neuron can be the data associated with a particular feature or the output of some other neuron. These input signals all have a weight that indicates how important it is in the context of the neuron. The computation within this neuron is then calculated by using a weighted sum, some bias, and an activation function. A weighted sum is calculated by using the following formula [9]:

$$Z = \sum_{i=1}^{n} (w_i \cdot x_i)$$

where n represents the number of input signals, w_i is the weight of an input signal i and x_i is the value of the input signal. If the weighted sum of input signals is 0, there is a bias introduced to the calculation. After this weighted

sum is calculated, the neuron applies an activation function to add non-linearity to the output. It is essential that a neural network is non-linear, as a strictly linear model would not be able to make predictions based on a more complex input.

Using strictly one pass of neurons would not be enough computation for most neural networks which is where the concept of layers is introduced. Layers allow the neural network to process data and learn. There are three main types of neural network layers: the input layer, hidden layers, and the output layer. The input layer takes the initial data as an input and performs the first round of neuron calculations. From there, the hidden layers perform most of the computation and are where most patterns are "learned." The output layer performs some final prediction or classification based on the type of problem that is currently being solved. In the case of binary classification, you may see a sigmoid classification, whereas for multiclass classification a SoftMax function may be applied.

The most basic way of learning involves some loss function which the machine learning model is attempting to minimize. In many cases, this is achieved by using gradient descent and calculating the mean squared error between the prediction output and the true output. Gradient descent lowers the coefficients of the prediction variables more slowly as time goes on to get the most precise value.

4.1.5 Convolutional Neural Networks (CNNs)

Many people view Convolutional Neural Networks as the step between trivial neural networks and graph neural networks as they possess qualities

of both. A common use case for CNNs lies in the context of images and their pixels. CNNs use filters to extract features from the image input. Feature extraction is the process of taking relevant information to make the computation required for the machine learning task simpler. After this, we can use what is known as a stride to observe the data around a pixel. The stride determines how many pixels away the filter moves after any operation.

Larger strides are more computationally efficient, but they fail to consider all the details of the pixels. Conversely, using smaller strides captures more details but requires an increase in computations. To model more complex relationships between features, CNNs use fully connected layers. This means that every neuron in one layer is connected to every neuron from the previous layer. By allowing information learned in previous layers to be preserved through layers, a CNN makes the most sense for an informed prediction in the case of image classification.

GNNs take the grid-like operations from CNNs and extend them to graphs. Their input is a graph data structure as is their output. An image can be represented as a graph with adjacent pixels being represented as neighbors. The features can be represented within a node by including information on the color. The most important operation in GNNs is message parsing.

Message parsing relies heavily on the concept of neighbors mentioned in Section 4.3.1. Through message parsing, nodes can communicate information on their neighbors through aggregation. The formula to represent message parsing is shown below [9].

$$h_v^{(l+1)} = \sum_{u \in N(v)} Agg^{(l)}(h_v^{(l)}, h_u^{(l)})$$

In this formula, $h_v^{(l)}$ represents node v in layer l. $N(v)$ represents the neighborhood of node, v and Agg represents the aggregation function used to combine the information from all the neighbors of v. Node v in layer l represents all the information about the node, including learned behaviors from previous layers. There are multiple aggregation functions, and common ones include mean pooling, sum pooling, or using attention. The aggregation function determines how the combination of all the neighbors' data is represented in just node v.

The layered architecture of GNNs updates the node based on the aggregation function. Since each layer of a GNN does message parsing and aggregation, all nodes would be updated with the most updated information at the end. Like strides in a CNN, graph theory introduces the idea of hops in the graph to explore depth. Hops are important to consider when deciding how message parsing and aggregation will take place. The most basic example used in GNNs is a social network, not an image grid. If we use this example to explain hops, one hop from node v would be looking at the friends of node v, whereas two hops would look at the friends of the friends of node v. This information gets aggregated into node v after two layers and now node v has knowledge from many nodes in the graph.

A benefit of GNNs is that they can quickly allow us to learn a lot about a graph, especially one that has high connectivity, though this may be very computationally expensive.

4.1.6 Graph Theory

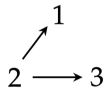
4.1.6.1 Graphs

Just like we learned about Neural Networks from our sponsor, we also need to know about Graphs for our project. Think of graphs as a bunch of dots (called nodes) connected by lines (called edges). These simple structures can show us how different things link up, like friends in a social network. When we mix this idea with Neural Networks, we get something super cool called Graph Neural Networks (GNNs), which are great for finding odd patterns. It's a combo that packs a punch, using both the strengths of graphs and neural networks.

A graph is a data structure designed to efficiently represent networks. In particular, a graph is a mathematical object, denoted $G = (V, E)$, which consists of a set of objects V, called *nodes*, and a relation $E \subseteq V \times V$, called the *edge relation*. By convention, an edge $(u, v) \in E$ represents a connection (e.g., arrow) from node u to node v. For each node u, its *neighborhood* $N(u)$ is the set of nodes that it is connected to, i.e., the set of nodes *incident* to u. Formally,

$$N(u) \coloneqq \{v \in V : (u, v) \in E\}.$$

As the above definition shows, a node need not be in its own neighborhood unless it is connected to itself. That is, $(u, u) \in E$. To illustrate these constructions, consider the following network:



The corresponding graph $G = (V, E)$ consists of nodes $V = \{1, 2, 3\}$ and edges $E = \{(2, 1), (2, 3)\}$. In this case, the neighborhoods are $N(1) = \emptyset$, $N(2) = \{1, 3\}$, and $N(3) = \emptyset$ where $\emptyset := \{\}$ is the empty set. Note that in the above network (and hence graph), the only connections between nodes are one-way channels.

In order for the graph to contain a two-way channel between nodes u and v, it needs to have both the edges (u, v) and (v, u). If every connection is a two-way channel, then the graph is said to be $undirected$, otherwise it is *directed*. Note that a graph being undirected is equivalent to saying that the edge relation E is *symmetric*. That is, for every edge $(u, v) \in E$, it must be true that $(v, u) \in E$. Formally,

$$\forall (u, v) \in E,\ (v, u) \in E.$$

As this example illustrates, with the right definitions, we can discover a lot of rich structures behind graphs. This motivates its formal study in mathematics called *Graph Theory*. We aim to utilize this theory to design an efficient algorithm for detecting network anomalies.

Graph theory is the study of graphs, which is a mathematical way to represent different types of data and the relationship between them. Graphs are composed of nodes and edges. Nodes are an abstract concept and can represent any sort of data or entity. Edges connect two nodes in a graph and may contain data as well. In this case, the IP address of one device can be a node, while the IP address of another can be the other node. The edge that connects these nodes could represent the payload sent between the two devices. Understanding what an attack looks like in terms of the data sent

between devices allows us to use graphs to make predictions on what type of attack is occurring.

Graphs are like a big web. Each dot or node can be anything - a person, a computer, or even a place. The lines or edges between them show how they are connected. These connections can go two ways (like a two-way street) or just one way. By visualizing these connections, it's easier to see how different things relate and interact with each other.

Graphs help our project a lot because they show data in a clear way. When we want to find something odd or different, seeing it on a graph makes it easier. It's a visual tool that gives clarity and can be super handy when dealing with a lot of info.

There are many types of graphs. Some show two-way connections (Undirected Graphs). Others show one-way connections (Directed Graphs). And some graphs use the lines to show how strong a connection is (Weighted Graphs). Each type has its unique way of showcasing relationships, giving us options to best represent our data. [10]

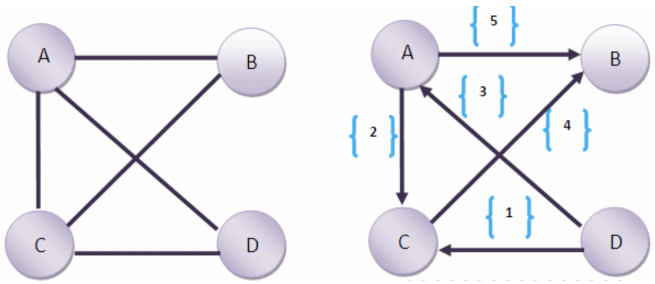


Figure 3. Image of an undirected (left) and directed graph with weighted edges(right).

Picking the right graph for our project is important. We need to look at our data and figure out which type of graph shows it best. It’s like choosing the best frame for a picture; the right choice makes everything clearer.

In tech, we can use graphs in many ways. Some smart tools, like what Google uses, figure out which websites are important by looking at their links. Others group similar dots together. It's a way to find patterns and trends in the web of information. By using these smart tools on our graphs, we can better find odd patterns. Knowing which parts of our graph are more important or which dots are similar can help our GNN work better. This combo helps speed up our search and make it more accurate.

To understand graphs, we use things like Degree (how many lines connect to a dot), Path Length (how far apart two dots are), and Clustering (how dots bunch up together). These measures help break down the graph's features and tell us more about its structure. Using these measures in our GNN can help it understand data better. For example, strange things might show up in dots with a lot of connections or on long paths. By keeping an eye on these measurements, we can refine our search for odd patterns.

Graphs are great, but they can have issues. They can get very big, which can slow things down. Also, in the real world, the dots and lines in a graph can keep changing. That means we always need to be on our toes and adjust as things change. We thought about these problems when making our GNN. Attempted to find ways to handle big graphs quickly and deal with changes to keep finding odd patterns accurately. Preparing for these challenges made our tool stronger and more reliable.

As the world gets more connected, from online friends to smart devices, graphs become more and more useful. They can help us understand big systems and find useful patterns. As things become more linked, the role of graphs will only grow. Our GNN project, built on graphs, is on the right track. Finding odd patterns in connected systems is becoming more important, so our work is very valuable. As the demand grows, our GNN could become a go-to tool for many.

Graphs, by showing how things connect, have a lot of uses on their own, but when we mix them with Neural Networks, we can find odd patterns even better. As we move forward, what we learned about graphs guided our project.

4.1.6.2 Neighborhoods and Connectivity

A primary concept of graphs that GNNs rely on is the neighborhood of graphs. The neighborhood of a node is the set of all its neighbors. We can learn a lot about nodes just by looking at their neighbors. In the context of networks, we can use a neighborhood to study the flow of traffic in a neighborhood and identify key nodes in the network. When making predictions on graphs, we can focus primarily on the nodes, edges, or the whole graph.

The only reason why we can look at the whole graph is the local connectivity of any node. As we explore the graph through a node’s breadth-first search, we are looking at the neighbors’ neighbors and we can associate the current breadth with how closely associated two nodes are. This is useful for community detection and node classification.

Bridges are a more complex concept of graphs that are also strongly related to anomaly detection. A bridge or cut-edge is an edge of the graph that, if removed, would result in an increase in the number of components in the graph. In some cases, this would be a strong indication of an anomaly as the node is only connected to the rest of the graph by one edge. Bridges are related to connectivity which refers to how strongly the nodes in a graph are connected to one another. A strong network would likely have relatively high connectivity. High connectivity suggests that the graph is resilient and indicates that there are only a few cut edges that, if removed, would impact the structure of the graph.

4.1.6.3 Adjacency Matrices and Lists

The standard way to encode graphs is as follows. Assuming the graph $G = (V, E)$ is finite (that is, there is only a finite number of nodes), we label the nodes in some arbitrary order $V = \{u_1, \ldots, u_n\}$. We then construct an *adjacency matrix* $A \in R^{n \times n}$ associated to the graph G where $A_{i,j} = 1$ if $\left(u_i, u_j\right) \in E$ and 0 otherwise. Consider the previous graph example. The nodes are already labeled from 1 to 3. Thus, its corresponding adjacency matrix is,

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

As a remark, note that a graph is undirected if and only if its adjacency matrix is symmetric along its main diagonal. Now with graphs encoded as matrices, we can feed the graph as input to a neural network. This is great if

we want to automatically learn the features of a single, fixed graph. However, some glaring issues naturally arise when we attempt to generalize to arbitrary graphs.

The most obvious obstacle is having a network that can process variable-sized graphs since the adjacency matrix grows quadratically with the number of nodes. Even if this were to be solved, another pressing obstacle is that the adjacency matrix is not unique given that a different ordering of the nodes would generate a permutation of the adjacency matrix. Such a neural network would need to be designed to be invariant under matrix permutation for it to be well-defined on graphs. And the last, more practical concern, is that adjacency matrices are very inefficient representations for graphs with few connections like trees or forests.

Instead of adjacency matrices, we can also encode graphs with *adjacency lists* which essentially associate each node with its neighborhood. This has the benefit of not requiring us to arbitrarily label nodes and hence leads to a unique representation for each graph without extraneous information. Unfortunately, it is not immediately clear how this representation can be fed into a neural network as now each node has a variable length array associated with it. To remedy this, let’s take a step back and see how graphs model communication networks in the first place.

4.1.7 Graph Neural Networks (GNNs)

Indeed, digital communication networks can be encoded as graphs where the servers are nodes and the open communication channels are

edges. If we associate to each node u a vector h_u, called a *node embedding*, we can then use h_u to represent packet data transmitted between servers.

Finally, if we allow h_u to be dependent on time k, denoted $h_u^{(k)}$, we can then use $h_u^{(k)}$ to represent the time evolution of a network. This extra structure to our graph defines a *dynamic graph on the nodes*. Similar embeddings can be made for edges $h_{(u,v)}^{(k)}$ and the graph $h_G^{(k)}$ as a whole which allows for even greater expressivity.

The procedure by which we transition from an embedding at time k to an embedding at time $k+1$ ultimately defines the behavior of our graph. To leverage the graph structure when transitioning between embeddings through time, we can restrict our transition function to only depend on local properties like current embedding and neighborhoods and global properties like graph embedding. In the most general setting, we iterate according to the following equations [11]:

$$h_{(u,v)}^{(k+1)} = update_{edge}\left(h_{(u,v)}^{(k)},\ h_u^{(k)}, h_v^{(k)}, h_G^{(k)}\right) \tag{1.1}$$

$$m_{N(u)} = aggregate_{node}\left(\left\{h_{(u,v)}^{(k+1)} : v \in N(u)\right\}\right) \tag{1.2}$$

$$h_u^{(k+1)} = update_{node}\left(h_u^{(k)},\ m_{N(u)}, h_G^{(k)}\right) \tag{1.3}$$

$$h_G^{(k+1)} = update_{graph}\left(h_G^{(k)}, \left\{h_u^{(k+1)} : u \in V\right\}, \left\{h_{(u,v)}^{(k+1)} : (u,v) \in E\right\}\right) \tag{1.4}$$

First, we update the edge embedding $h_{(u,v)}^{(k+1)}$ using its past embedding, the embedding of its incident nodes, and the graph embedding (Eq. 1.1). Next, we update the node embedding $h_u^{(k+1)}$ using its past embedding, an aggregate $m_{N(u)}$ of all embeddings of its incident edges, and the graph embedding (Eq. 1.2 and 1.3). Finally, we update the graph embedding $h_G^{(k+1)}$ using its past embedding, the embedding of all nodes, and the embedding of all edges (Eq. 1.4). Note that the update and aggregate functions are arbitrary. To illustrate how the transition procedure works, consider Figure 3.

The diagram in Figure 3. illustrates where the embedding $h_\star$ of the top-left node $\star$ traverses as we move forward in time (i.e., layers). In particular, the top left node is adjacent to the top and top-right nodes so $h_\star$ moves to those respective nodes on the first iteration (layer 1). From there, the top node is adjacent to all other nodes in the graph. Thus $h_\star$ moves to all nodes. [12]

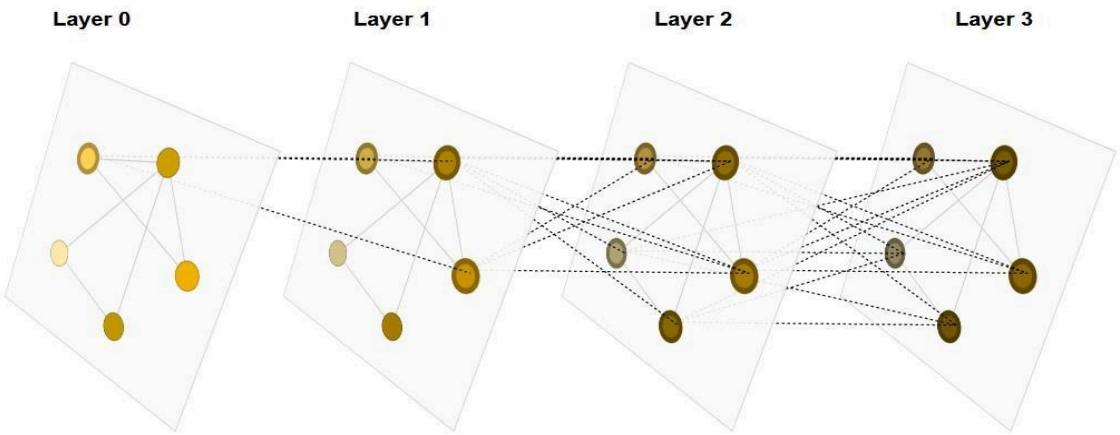


Figure 4. Diagram of embeddings passing through an undirected graph in three iterations [12].

Other than the top node on the second iteration (layer 2). However, the top-right node is adjacent to the top-left and top nodes. Thus $h_\star$ also moves to the top node on the second iteration. Therefore, $h_\star$ is considered in every node of the graph at the second iteration. A similar argument applies to the third iteration (layer 3).

In Figure 3, we only consider where the embeddings go with each iteration. However, if we apply some computation step in the middle, we can begin to calculate the properties of the graph. In particular, with the appropriate choice of update and aggregate functions, we can leverage these graph dynamics to compute network features at the node level, edge level, and graph level. How to select these update and aggregate functions depends on the problem but with a suitable parameterized construction, we can apply the same techniques as in neural networks to learn the functions.

The above construction defines a *graph neural network* (GNN). The reason for this terminology is that the transition functions for graphs parallel to the transition functions for neural networks. In fact, graph neural networks can be thought of as a generalization of convolutional neural networks (CNN) since the corresponding graph for CNNs has nodes representing pixels with edges connecting adjacent pixels [9].

Graph neural networks go through data from each node, it finds relationships by looking at the edges. The layers of the network are equal to the diameter of the graph. Each layer will look past another edge. This makes it so that each layer has more information than the last, making it such that the last layer will have all the relationships that occur in the graph. Because a graph neural network will show the ideal relationships in a graph, an anomaly will be a large outlier in the graph making it easy and efficient to detect.

Figure 2 shows the representation of the types of graph neural networks that currently exist. One of the big decisions that we must make is to see if we should use a static or dynamic graph. The difference is that a static graph does not change over time, making the computation easier and the neural network easier. Dynamic graphs on the other hand are a graph that changes over time. This is relevant to networks as networks are constantly evolving, there are many devices entering and leaving the network. Dynamic graphs are also the least researched graph type, and the only type of detection that currently is written about is anomalous node detection and anomalous edge detection.

After looking through this we realized that we must first understand the graph neural network frameworks to find out what types of graph neural

networks exist. The simplest framework is a graph convolutional network. A graph convolutional network operates by convolving filters on a graph, letting it extract patterns and relationships in the network. It is like convolutional neural networks but applied to graphs. There is then the network architecture that you want to use to find anomalies. There are many architectures that can be used, usually the researchers create their new novel idea with these methods.

Graph Neural Networks combine the graph data structure with neural networks to enhance data analysis and machine learning. The main reason why this project is suited for a GNN architecture is because not only do we need to look at individual packets but also the connections between devices. Using a graph gives us an understanding of the whole network. To have a grasp on how GNNs operate, a basic understanding of graph theory is essential.

After diving into both Graphs and Neural Networks, let's bring them together to understand Graph Neural Networks or GNNs. GNN’s combine both the web-like structure of graphs, and the human brain-like aspect of Neural Networks. They’re great for cyber security tasks like ours of detecting anomalies in data, and are designed to find odd patterns and details in a special way.

GNNs use the best parts of graphs and neural networks. They understand the nodes and connections/edges of graph data. For our project, GNNs are a game-changer. They let us see odd patterns in data that’s all linked up. We're not just looking at random bits of info; we're diving deep into how everything is connected. [12]

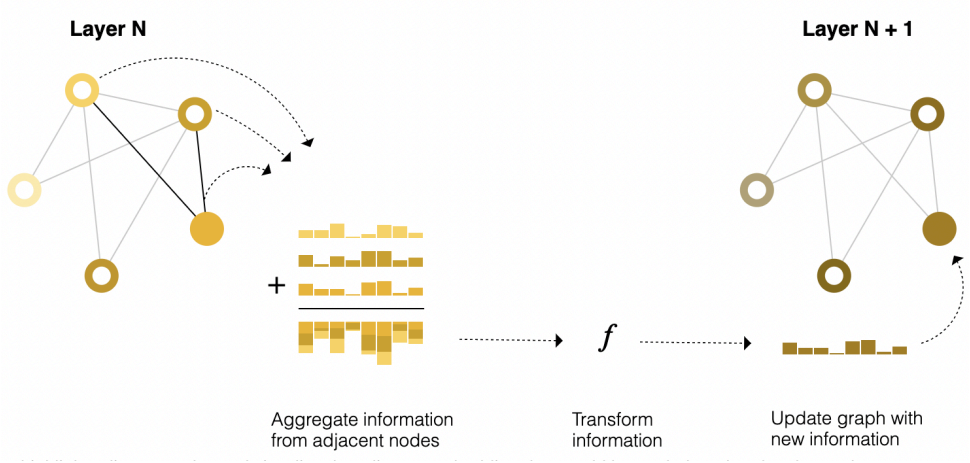


Figure 5. Illustrates the message-passing operation in the GNN. [12]

Like neural networks, GNNs learn from data. But they focus on the links and connections between things, this leads to GNNs being able to get better and better after they are built. This is because they make smart guesses on connections by going back and forth over the data being passed through them.

This unique way of learning means GNNs are great for spotting hidden patterns in connected data. One of their strengths is understanding the web-like patterns of the data, and passing it through this structure, which is perfect for our goal of finding odd patterns or in our goal's case, anomalies.

Even though GNNs are powerful and effective in many scenarios, there are still times where they have obstacles to get through. They need lots of data to learn well, and sometimes, they can take a long time to process all that data, especially when the links and connections get complicated.

By combining graphs and neural networks, GNNs offer an efficient process that is able to spot odd patterns effectively. As we continued through our project, the lessons from both were our foundational guidance.

It is important to know the different types of GNNs as well as their strengths/weaknesses. We're working to make Graph Neural Networks (GNNs) even better. We can improve how they see and understand data, make them adjust to new information faster, and help them manage bigger sets of data more easily. Combining GNNs with other tech models could help spot issues more widely. Plus, if we let our GNN learn from its past actions, it can get smarter over time. All these steps make our tool stronger and more prepared for future challenges.

Graph Neural Networks (GNNs) excel at learning patterns and relationships within graph-structured data. The extracted features, including source and destination IP addresses, port numbers, protocol information, and timing characteristics, can serve as node and edge attributes in the graph representation. These features encapsulate the contextual information necessary for GNNs to discern normal from anomalous network behavior.

In conclusion, the featurization of packet capture files using PyShark, coupled with the power of Graph Neural Networks, offers a robust approach to anomaly detection in network traffic. The extracted features provide rich contextual information, and the graph structure allows for a nuanced understanding of network interactions. This methodology contributes to the advancement of network security practices, enabling more accurate and proactive identification of anomalies.

4.1.8 Tools for Graph Neural Networks

4.1.8.1 Connectedness Ratio

When studying graphs, an important characteristic to look for is how connected the graph is. On first-order considerations, we only look at what nodes each node is connected to. This is called the node's neighborhood as discussed before. On the next stage, we may want to look at how connected the neighborhood is. That is, checking if each neighboring node is connected to each other. Intuitively, the more connected a node's neighborhood is, the more the neighborhood acts like a single node.

We can quantify this numerically by taking a ratio of the connectedness of the neighborhood compared to the maximal scenario where every node in the neighborhood is connected to each other. Symbolically,

$$c_u = \frac{|(v_1, v_2) \in \mathcal{E} \ : \ v_1, v_2 \in \mathcal{N}(u)|}{\binom{d_u}{2}}. \qquad [13]$$

Here, c_u is the connectedness ratio of a node u which is calculated as the ratio of the number of edges $(v_1, v_2) \in E$ within the neighborhood $N(u)$ treated as a subgraph and the maximal possible case d_u choose 2 where d_u is the number of nodes in the neighborhood of u.

The usefulness of the connectedness ratio is that it allows us to give a measure for how fully connected a neighborhood is. As stated above, a neighborhood which is close to fully connected should act more like a single node at least in the context of communication networks. This is because they all influence each other in a significant / direct way. Hence any edges going outside the neighborhood will likely be dependent on the activity of the entire neighborhood, not just a single node.

4.1.8.2 Basic Overlap Statistic

We may also care about the relationship between two neighborhoods. The simplest statistic on this is a first-order consideration which is the basic overlap statistic which is a count of nodes shared between two neighborhoods. Symbolically, [13]

$$\mathbf{S}[u, v] = |\mathcal{N}(u) \cap \mathcal{N}(v)|, \quad [13]$$

Here, $S[u, v]$ is the number of nodes within the neighborhood $N(u)$ of node u and neighborhood $N(v)$ of node v. As an aside, knowing the size of the neighborhoods $N(u)$ and $N(v)$, in addition to the basic overlap statistic, gives us enough information to calculate the size of $N(u) \cup N(v)$ which may be useful for analyzing the influence two nodes have on the network as a whole. Importantly, the basic overlap statistic is proportional to the probability that the two nodes are adjacent to each other. Symbolically, [13]

$$P(\mathbf{A}[u, v] = 1) \propto \mathbf{S}[u, v]. \quad [13]$$

Keep in mind that the proportionality is with respect to change in nodes u and v. With the basic overlap statistic, one can study whether two nodes share influence over a lot of nodes in quantity. This is useful when one cares about the scale by which two nodes influence the graph and less so whether the two nodes have the same topology.

4.1.8.3 Sorenson Overlap Statistic

That said, the basic overlap statistic doesn't take into consideration how connected the two neighborhoods are. Specifically, if both nodes u and v have a high degree in the graph with low overlap, the basic overlap statistic would ignore that their neighborhood overlap is practically nonexistent, compared to one where a majority of the neighborhood overlaps.

We can remedy this by taking a proportion of the overlap with respect to the total number of nodes in each neighborhood. Specifically this is called the Sorenson overlap statistic. Symbolically, [13]

$$\mathbf{S}_{\text{Sorenson}}[u, v] = \frac{2|\mathcal{N}(u) \cap \mathcal{N}(v)|}{d_u + d_v}, \quad [13]$$

In contrast to the basic overlap statistic, the Sorenson overlap statistic allows us to study whether two nodes have a high proportion of overlap. If

they do, then both nodes can be thought of as having equal contribution to the graph topology and dynamics.

Specifically, when the Sorenson overlap statistic is close to 1, both neighborhoods are practically the same. Indeed, at the limit of 1, the neighborhoods are exactly the same. If the Sorenson overlap statistic is close to 0, both neighborhoods are practically disjoint. Indeed, at the limit of 0, the neighborhoods have no common node.

This is especially useful for network analysis when attempting to identify if two nodes share similar influence on the network. Pairs of nodes with high Sorenson overlap statistic would imply that we should treat the two nodes as practically the same node. This gives a more refined analysis than the connectedness ratio since we are looking at a pair of nodes instead of an entire neighborhood.

4.1.8.4 Katz Index

As mentioned, the Sorenson overlap statistic may be used to give a rough estimate on the similarity in influence two nodes have over a graph. We can get a better estimate by extending our neighborhood consideration into larger depths. Intuitively, the further away a node is connected to another, the less influence they have over each other. To quantify this, we can scale by a decaying factor with respect to the neighborhood depth. This is called the Katz index. Symbolically, [13]

$$\mathbf{S}_{\text{Katz}}[u, v] = \sum_{i=1}^{\infty} \beta^i \mathbf{A}^i[u, v], \quad [13]$$

Here, A^i is the ith-step adjacency matrix where $A^i[u, v] = 1$ if and only if there is a path of exactly length i between nodes u and v. Moreover, β^i is the decaying factor where β is chosen such that it is smaller than the reciprocal of the absolute value of the largest eigenvalue of A. The Katz index gives a measure of how tightly connected nodes u and v are.

However, much like the basic overlap statistic, the Katz index suffers under the heavy influence of nodes with high degree. This is partly because a node with high degree gives a lot more opportunities for a path of length i to exist between two nodes. In particular, it would artificially blow up the adjacency matrix count compared to a node which is just as strongly connected but which does not have as many opportunities to branch. To remedy this we can take the expected path value between two nodes. [13]

$$\mathbb{E}[\mathbf{A}[u, v]] = \frac{d_u d_v}{2m}, \quad [13]$$

We then invert the expected path value and use it to scale the Katz index. This allows us to take into account nodes with high degree since it acts as a normalizing constant, scaling down the count on path lengths which are more likely to carry an instance from u to v. That said, the Katz Index is very useful in giving an estimate on just how strongly connected

two nodes are to each other. Thus any analysis on the indirect influence a node has on another can find fruitful use with the Katz Index.

4.1.8.5 Katz Centrality

The previous measures all look at how related two nodes are based on their influence over a graph. We can use these measures to assign a quantity on each node that measures their respective influence over the entire graph. More specifically, we can get a measure of the relative influence of a node within a graph by fixing a node u and summing a given measure over all nodes v in the graph.

If we perform this procedure with the Katz Index in particular, we get the Katz centrality of node u. Intuitively, the Katz centrality measures the relative influence of a node within a graph since it sums up the influence a node has on each node in the graph. If we take the normalized Katz Index based on expected path length, then we can normalize by the size of the graph to get a proportion on how strongly a node influences the graph.

By influence, we mean whether a path exists from a node to another node and by how much. Clearly for a complete graph, every node influences the other nodes by the same amount, and hence the proportion should be the same for all nodes. This is indeed what happens when using the Katz centrality measure.

In contrast, a graph where a select few nodes have direct connections to almost the entire graph would have a much higher Katz centrality measure than the other nodes which are connected indirectly to the rest of the graph.

4.1.8.6 Laplacians

When dealing with dynamical systems in physics, many important characteristics of the systems are dependent on how far the object "diverges" from the "average". For example, when studying gasses, it is important to know how gasses spread, i.e. diffuse. This is generally calculated using the Laplacian, or the Laplace operator [14].

In classical mechanics, the Laplacian is a differential operator that is given by the divergence of a conservative field and hence measures how much a point is a source or sink in the field. In differential geometry, the Laplacian has a more geometric interpretation. It measures the curvature of a field and, in particular, tells you how far the value of a point deviates from its neighboring points in the space.

We can adapt this concept to discrete structures like graphs. In particular, the Laplacian as a differential operator can be approximated using a finite-difference equation. This gives the discrete Laplacian. Most importantly, the finite-difference equation essentially samples values from its neighbors and applies some weighted average. We can generalize this by extending our definition of neighbors to that of graph neighbors and extend the weighted average in a natural way. The result is the unnormalized Laplacian [13]

$$\mathbf{L} = \mathbf{D} - \mathbf{A}, \quad [13]$$

which is defined as the difference between the degree matrix, a matrix where the diagonal consists of the degree of each node, and the adjacency

matrix. This acts much like a finite-difference approximation of a derivative but where our geometry, instead of being euclidean space, is a graph.

However, as presented, the unnormalized Laplacian is scaled by the maximum degree in the graph which leads to scaling issues in calculations. By normalizing to a ratio, we get a unique operator associated with a given graph that tells us, by proportion, how each node relates to other nodes in terms of connectivity. In particular, we get the normalized Laplacian: [13]

$$\mathbf{L}_{\text{sym}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{L}\mathbf{D}^{-\frac{1}{2}}, \text{ [13]}$$

This inverts the Unnormalized Laplacian by the degree, turning the matrix into a ratio. Observe that since the Laplacian matrix is a linear operator, it corresponds to a linear transformation from graph nodes to some spectral domain. The corresponding spectral decomposition of a graph via the Laplacian can be used to construct low-dimensional embeddings of the graph for purposes of machine learning. The eigenvectors of the Laplacian matrices moreover can be used to create an ideal clustering of graphs.

4.1.8.7 Cut

At times, it may make sense to observe just how concentrated a graph is in terms of forming separate cliques. To quantify just how clustered a graph is, we can define a list of subgraphs of interest and measure how far these subgraphs extend beyond their set of nodes. In particular, we can count how many edges leave each subgraph and add them up to give a score. Symbolically, [13]

$$\text{cut}(\mathcal{A}_1, ..., \mathcal{A}_K) = \frac{1}{2}\sum_{k=1}^{K} |(u, v) \in \mathcal{E} \ : \ u \in \mathcal{A}_k, v \in \bar{\mathcal{A}}_k|.$$

[13]

The score assigned to this set of subgraphs then tells us how far away the graph is in being clustered into these subgraphs. In particular, a low cut signifies that our graph is mostly contained in one of these subgraphs and is mostly disjoint from the others. A high cut signifies that our graph is strongly connected between subgraphs and that there is not as much clustering.

The usefulness of this measure is that we can give a quantity for how isolated a feature is in the graph where a feature defines a subset of nodes of interest. If the feature is very isolated (that is, the cut is low), then there is likely a linear behavior in assignment of this feature. If in contrast, the feature is not very isolated, then there may be strong nonlinear behavior influencing the feature's assignment.

4.1.8.8 Ratio Cut

Of course, like before, the cut measure is heavily weighted by the degree of the nodes within the subgraph. That is, a subgraph with a high cut score but even higher degree within the subgraph itself may wrongly be interpreted as saying that the subgraph is not strongly independent. This is not ideal, since a large complete subgraph is very much an independent unit of the graph if the outwards edges are miniscule in comparison to the graph. To remedy this issue, we normalize! Symbolically, [13]

$$\text{RatioCut}(\mathcal{A}_1, ..., \mathcal{A}_K) = \frac{1}{2}\sum_{k=1}^{K} \frac{|(u,v) \in \mathcal{E} \ : \ u \in \mathcal{A}_k, v \in \bar{\mathcal{A}}_k|}{|\mathcal{A}_k|}, \quad [13]$$

This computation involves determining the ratio of outgoing edges from a given subgraph to the total number of nodes within that subgraph. Essentially, this ratio reflects the degree of connectivity among the nodes within the subgraph and serves as a meaningful indicator of the subgraph's internal cohesion relative to its external connections within the larger graph.

Remarkably, the pursuit of an optimal collection of subgraphs, minimizing the ratio cut, leads us to explore the spectral decomposition associated with the second smallest eigenvalue of the Laplacian matrix. This analytical approach provides a systematic means of identifying subgraphs that efficiently partition the graph into distinct, largely independent clusters.

By leveraging the spectral decomposition and focusing on the second smallest eigenvalue, we can algorithmically pinpoint subgraphs that foster well-defined and isolated groupings within the broader graph structure.

4.1.8.9 Volume Cut

That said, the ratio cut may seem rather unintuitive to be considered a ratio. This is because it measures how much a subgraph is connected relative to the entire graph rather than how much a subgraph is connected

relative to itself. In particular, the maximal number of edges within a subgraph grows quadratically with respect to the number of nodes.

Thus, we can find a pathological example, where we have a large complete subgraph with a single node that has out degree linear with respect to the node count so that the ratio cut gives us a high ratio even though the subgraph should very much be treated as independent given that it has significantly more connections within the subgraph than it does outside.

To remedy this issue, we can alter our ratio to be with respect to the number of edges within the subgraph instead of just the number of nodes. Notationally, we call this the volume of the subgraph denoted $vol(A_k)$. Symbolically, [13]

$$\mathrm{NCut}(\mathcal{A}_1, ..., \mathcal{A}_K) = \frac{1}{2}\sum_{k=1}^{K} \frac{|(u,v) \in \mathcal{E} \ : \ u \in \mathcal{A}_k, v \in \bar{\mathcal{A}}_k|}{\mathrm{vol}(\mathcal{A}_k)}, \qquad [13]$$

This formula thus gives a measure of how much a subgraph is connected to itself compared to outside. In particular, the ratio is less than 1 if the subgraph is mostly connected to nodes within the graph. The ratio is greater than 1 if the subgraph is mostly connected to nodes outside the graph.

As a result, the total volume cut gives a measure of how much the subgraphs are completely isolated from the rest of the graph. This is an especially useful measure for cliques since we already know the subgraphs

are complete but we would like to know whether the subgraph has greater influence outside of its set than to itself.

4.1.8.10 Encoder-Decoder

With all the measures of a graph now defined, it is time to give a framework in which we can learn features of the graph using standard machine learning techniques. In particular, a graph by itself is rather difficult to study. The simplest way to overcome this problem is to encode the nodes (and/or edges) of the graph into some ambient euclidean space where the encoding is chosen so that it is possible to extrapolate the graph structure from the embeddings.

We can describe this mathematically as an encoder-decoder model. Particularly, the encoder assigns each node an embedding in the ambient space. The decoder seeks to extrapolate some topological information between two nodes by just looking at their corresponding embeddings. Symbolically,

$$\mathrm{ENC} : \mathcal{V} \to \mathbb{R}^d,$$ [13]

$$\mathrm{DEC} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+.$$ [13]

We can think of the decoder as a pseudo-metric function that gives a measure of how strongly correlated two vectors are under a given topological consideration. This is not a true metric, or at least it shouldn’t be

thought of as such since the graphs we are dealing with are finite and hence the topology induced by the metric is guaranteed to be discrete.

By aligning the decoder to gauge the similarity score between two nodes based on a specific graph consideration, we achieve a successful encoding of the graph into an ambient vector space. This alignment allows us to effectively discriminate between nodes, considering the nuances of the graph's underlying structure. In this manner, the decoder serves as a valuable tool for navigating and interpreting the intricate relationships within the graph, facilitating the extraction of meaningful insights from the encoded vector space.

4.1.8.11 Loss Function

To achieve the goal set before, we can attempt to learn the encoder-decoder functions by identifying them in some parameterized family and then applying some loss function that measures the error in the estimate between the decoder and the similarity measure. In particular, the total loss is quite simply the sum of the loss across each pair of nodes in the graph. Symbolically, [13]

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} \ell\left(\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v), \mathbf{S}[u,v]\right), \quad [13]$$

The loss function could be any classification function but should be chosen so that the overall function can be thought of as smooth with respect

to the embeddings (and the similarity measure fixed). Defining our encoder-decoder models as a neural network, the total loss function is then a real valued function that can be subjected to backpropagation which is then used to optimize the neural networks.

This is especially powerful in that it allows us to train a neural network to encode a graph into a low-dimensional euclidean space. With appropriate loss and similarity measure, we can construct an encoding so that average network activity is bounded within some hypersphere. The anomalous nodes in the network will then be identified as those nodes which are encoded outside the hypersphere threshold. In fact, this model is the primary method in which state-of-the-art graph neural networks detect anomalous nodes within a network either communication, financial, or social.

4.1.8.12 Tensor Decomposition

Of the many mathematical tools deployed in machine learning tools, tensors, has been one of the more successful additions, especially at handling deep networks and large dimensionality. This is partly because tensors offer a compact representation of multi-dimensional data with symmetry properties that make it very nice to work with in contrast to regular matrices. The biggest utility comes from tensor decomposition [15].

In contrast to matrix decomposition, which often demands stringent conditions for a unique factorization, tensor decomposition operates under more lenient prerequisites, leveraging higher dimensions to introduce additional symmetries that enforce uniqueness in the factorization process.

The inherent structural complexity of tensors, facilitated by these symmetries, guarantees unique factorization. This unique characteristic sets tensor decomposition apart, offering a more forgiving framework where a one-of-a-kind factorization can be attained under weaker conditions. The nuanced interplay of symmetries within tensors not only simplifies the factorization process but also underscores their versatility in handling complex data structures, positioning tensor decomposition as a powerful tool in diverse fields, from data science to signal processing and beyond.

Computationally, the main interest in constructing a decomposition is because we can transform a high-dimensional tensor into either a product or sum of low-dimensional tensors. That is, we can reduce the dimension of our data which makes it easier to analyze with statistical methods and also work with in a machine learning context.

Ultimately, in the context of graph neural networks, tensors provide a means for us to store not only structural graph data but also temporal data as an extra dimension. This allows us to then construct a model that considers network activity through time instead of just a single instance of time which is essential for identifying DDoS attacks or timing attacks.

4.1.8.13 Adjacency Tensor

However, a temporal dimension is not the only thing a tensor can be used for. In fact, where the adjacency matrix serves as a linear algebraic representation of graphs, adjacency tensors serve as a multi-linear algebraic representation of hypergraphs [16].

In particular, a hypergraph is one in which edges can connect to more than just a pair of nodes. Mathematically, this can be thought of as generalizing the edge set to simply being a family of subsets of the vertex set with no restriction on the subsets being at most cardinality two.

However, we still want to be able to have some structure in the hypergraph, so we instead consider k-uniform hypergraphs where every edge set has at most cardinality k. A regular graph is then a 2-uniform hypergraph. Representing these hypergraphs can no longer be done perfectly with a simple adjacency matrix since an adjacency matrix only looks at pairwise connections. To go around this, we introduce the notion of an adjacency tensor.

Much like a regular adjacency matrix, the adjacency tensor has k-degrees of freedom where each degree has dimension equal to the number of nodes in the graph. An entry in the adjacency tensor is one if and only if the corresponding set consisting of nodes corresponding to the entry-indices is an edge in the hypergraph.

The utility of this construction for network anomaly detection is that communication networks may be laid out so that a single node is communicating the same message to multiple nodes. This occurs especially in distributed systems or power grids where a single plant supplies power to multiple units. The simplest way to encode these networks is as a graph where an edge exists if there is a path between the nodes in the underlying network. The issue with this approach is that it divides a distributed network connection into multiple pairwise connections that have no correlation in the graph. That is, it loses information that the edges are actually related to each other.

To remedy this, we can instead encode the network as a hypergraph so that the distributed network connection is in fact one edge. A directed flavor to the hypergraph can also be imposed by requiring that the first index represents the source of the edge and all other indices represent sinks of the edge. Of course, the corresponding adjacency representation is fundamentally multi-dimensional which is why we use adjacency tensors instead of adjacency matrices.

4.1.8.14 Multi-Relational Data and Knowledge Graphs

Multi-Relational graphs are defined by their variance in the types of edges that can exist between two nodes.

$$e = (u, \tau, v)$$ [13]

e represents an edge in the graph, u and v indicate what nodes are being connected by the edge, and τ, represents the type of edge. Because of the introduction of different edges the decoder function works differently

$$\mathrm{DEC}(u, \tau, v) = \mathbf{z}_u^\top \mathbf{R}_\tau \mathbf{z}_v, \quad [13]$$

Instead of taking in node embeddings it also takes in a relation type where R_T is a matrix that represents the relationship between the different

nodes and Z_u and Z_v represent the node embeddings of two connected nodes. R_T is trained using backpropagation. The issue with this model is that it works in $O(d^2)$ as opposed to latter models that work in O(d)

4.1.9 Anomaly Detection

Due to the complex nature of GNNs, anomaly detection involves determining the best method for the specific use case. Graphs can be static or dynamic, dynamic graphs contain connections that change over time, making anomaly detection a complicated task. Alongside this, anomalies can be detected at the node, edge, and graph level. [1]

Graph type	Anomaly type	Network architecture	Method	Summary (key issue addressed → solution)
Static graph	Node anomaly	GCN-based GAE	DOMINANT [7]	Complex interactions, sparsity, non-linearity → GCN-based encoder
			Dual-SVDAE [21]	Overfitting for normal & abnormal → hypersphere embedding space
			GUIDE [22]	Complex interactions → higher-order structure decoder
			SpecAE [8]	Over-smoothing issue → tailored embedding space
			ComGA [23]	Over-smoothing issue → community-specific representation
			ALARM [24]	Heterogeneous attributes → multiple GCN-based encoders
			AnomMAN [25]	Heterogeneous attributes → multiple GCN-based encoders
			SL-GAD [26]	Contextual information → subgraph sampling & contrastive learning
		GCN alone	Semi-GCN [27]	Label information → semi-supervised learning by GCN
			HCM [28]	Label & contextual information → hop-count prediction model
			ResGCN [29]	Over-smoothing issue → GCN with residual-based attention
			CoLA [30]	Targeting issue of GAE → contrastive self-supervised learning
			ANEMONE [31]	Contextual information → multi-scale contrastive learning
			PAMFUL [32]	Contextual information → pattern mining algorithm with GCN
		GAT-based GAE	AnomalyDAE [33]	Complex interactions → GAT-based encoder
			GATAE [34]	Over-smoothing issue → GAT-based encoder
			AEGIS [35]	Handling unseen nodes → generative adversarial learning with GAE
		Other GNN-based model	OCGNN [36]	Targeting issue of GAE → GNN with hypersphere embedding space
			AAGNN [37]	Targeting issue of GAE → GNN with hypersphere embedding space
			Meta-GDN [38]	Hard work to label anomalies → meta-learning with auxiliary graphs
	Edge anomaly	GCN-based GAE	AANE [39]	Noise or adversarial links → GAE with a loss for anomalous links
			eFraudCom [40]	Fraud detection → heterogeneous graph and representative data sampling
		GCN alone	SubGNN [41]	Fraud detection → GIN and extracting and relabeling subgraphs
	Subgraph anomaly	GCN alone	GLocalKD [42]	Graph-level anomalies → joint learning global & local normality
		GAT-based GAE	HO-GAT [43]	Abnormal subgraphs → hybrid-order attention with motif instances
	Graph-level anomaly	GCN alone	OCGIN [44]	Graph-level anomaly detection → graph classification with GIN
			OCGTL [45]	Hypersphere collapse → set of GNNs for embedding
Dynamic graph	Edge anomaly	GCN and GRU	AddGraph [10]	Long-term patterns → temporal GCN with attention-based GRU
			DynAD [46]	Long-term patterns → temporal GCN with attention-based GRU
			Hierarchical-GCN [47]	Dynamic data evaluation → temporal & hierarchical GCN
			StrGNN [48]	Structural change → mining unusual temporal subgraph structures
	Node anomaly	GCN & DRNN-based GAE	H-VGRAE [49]	Anomalous nodes → modeling stochasticity and multi-scale ST dependency
		GCN and GRU	DEGCN [50]	To capture node- and global-level patterns → DGCN and GGRU
		GCN alone	TDG with GCN [51]	Malicious connections on traffic → extracting TDGs

Figure 6. Diagram to determine which method to use for GNN-based anomaly detection. [1]

As see in the image above, most node anomaly GNN-based approaches for detecting anomalies are based on the Graph Autoencoder (GAE) framework. The GAE framework encodes the node's features into a representation of a lower dimension. By reducing the dimensions of the node while keeping important features, there is faster computation. After this the decoder recreates the graph structure. From here the general neural network architecture is followed through a loss function and training followed by evaluation.

Another common acronym seen in the diagram is GCN which stands for Graph Convolutional Network. A GCN is a specific type of GNN that uses forward propagation for convolution operations. Extensions of them explore isomorphisms within graphs and attention.

Anomaly detection is the way of identifying abnormalities in the graph which deviate from the norm. The four types of anomalous detection are anomalous node detection, anomalous edge detection, anomalous subgraph detection and graph level anomaly detection.

The anomalous node detection detects if a node is anomalous. This is useful for our project as this is a straightforward way to detect if there is a compromised or malicious machine on the network. Anomalous edge detection is useful for our project as it is what you can use to find malicious packets in the network. Subgraph detection is a more complicated way to detect anomalies, it uses the node detection and motifs to find if there is an anomaly. Graph level anomaly detection detects graphs inside the graph that are anomalous.

The state of the art for graph neural networks are static or dynamic graphs, different types of anomalous node detection. This is also the most

researched type of anomalous detection, which can make it easier for us to build. Trying to create a dynamic graph that detects anomalies using any of the four ways is a large achievement for an undergraduate student group. The simplest way for us to make a novel way to solve anomaly detection is to create a method under these four types of detection.

The first framework that we researched was the graph convolutional network-based graph autoencoder (GAE). This graph changes the decoder depending on the perspective of the method. This is often used when there is a non-linear complex relationship between the nodes. This identifies the anomaly by scoring the differences in the basis of reconstruction errors. The autoencoders have dual encoder-decoder architecture.

Graph convolutional networks are used when the data should be run with unsupervised learning. These are very similar to convolutional networks, as they are localized to their neighbors. This is used to predict interactions rather than the individual. One of the problems with graph neural networks is that there is no inherent order in the nodes, all that matters is the layer and how many edges you go through to see the neighborhood.

In order to make meaningful progress on this problem, we needed to first know the current methods that have been developed in the domain of detecting anomalies in digital communication networks. Fortunately, there is a paper that surveys and categorizes most of the methods that have been developed over the past couple of years related to the subject, although in a more general setting of arbitrary abnormality detection.

As we saw in the table, there are two major problem domains: static graphs and dynamic graphs. The former refers to algorithms which perform

anomaly detection on snapshots of networks. The latter refers to algorithms which consider the time evolution of the network. For both of these, the most approachable anomalies are node and edge-level detection. The more difficult being subgraph and graph-level detection.

For the most part, state-of-the-art anomaly detection algorithms use the simpler GNN model that only employs node embeddings. That is, they primarily work off the following equations:

$$m_{N(u)} = aggregate^{(k)}_{node}\left(\left\{h_v^{(k)} : v \in N(u) \cup \{u\}\right\}\right) \text{ [13]}$$

$$h_u^{(k+1)} = update^{(k)}_{node}\left(h_u^{(k)}, m_{N(u)}\right) \text{ [13]}$$

Note, for generality, these models may have aggregate and update functions that change with each iteration as well. This allows the GNN to be split into stages, much like a standard convolutional neural network where there are layers for convolution / encoding, padding, and decoding.

The general attack strategy to solve the network anomaly detection problem involves taking an attributed graph, processing it through a GNN to extract relevant features, then measuring its distance from some set standard. That is, we take in a graph with some prespecified embeddings, run it through a GNN for some fixed number of iterations (where the update function is learned via backpropagation), identify some threshold on the resultant embeddings which defines the model's confidence in the nodes

exhibiting anomalous behavior, and finally measure distance from this threshold to classify nodes.

A representative state-of-the-art model that uses this technique is the *abnormality-aware graph neural network* (AAGNN) [17]. This model uses community behavior as the background that defines normal behavior. Anomalous node behavior, then, is defined as those nodes which deviate from the behavior of their neighbors. By deviation, we mean the following. Nodes having high confidence in representing average behavior in their neighborhood are used as labels to learn a hypersphere boundary within the latent feature space produced by an autoencoding GNN procedure. Nodes that fall outside this hypersphere boundary are considered anomalous.

For the case of solving the more difficult problem of subgraph anomaly detection, a representative state-of-the-art model that uses the autoencoding technique is the *hybrid-order graph attention network* (HO-GAT) [18]. This model uses an autoencoding GNN procedure to learn suitable node representations and motif instance representations of a given graph. The node representation focuses on the fine-grained details of a graph while the motif instance representation focuses on connectivity patterns within the graph. The combination of these two gives a representation that allows analyzing properties both at the node-level and subgraph-level. Finally, the reconstruction error is used as the abnormality score of nodes and motif instances respectively.

As of now, anomalous behavior is defined in terms of what nodes are acting outside the norm of the network. With the right embedding or attack focus, this would correspond to the malicious actors themselves. However, this is not true in general and may instead just identify the symptoms of an

attack. A particular example is DDoS attacks where botnets can be used to target a single node in waves without the sources acting anomalous themselves.

This problem is addressed via subgraph-level detection. However, as mentioned before, the research on subgraph anomaly detection is limited and requires expensive computation to construct motifs, multi-stage graphs, and other structures that can optimally be used to analyze subgraphs. We offer an alternative approach inspired by backpropagation.

The main idea is to identify abnormal nodes and then backpropagate the abnormality to find nodes which contributed to the behavior. This is trivial for attacks involving only one malicious actor. However, for malicious subgraphs, note that such subgraphs generally induce abnormal behavior by acting in unison within some sufficiently small window of time (i.e., like a hive mind).

As such, if we attempt to backpropagate, the contributions of all the nodes in the subgraph should be around the same. With this assumption, we should be able to recover the full subgraph that contributed to the abnormality. With repeated applications on a sliding time window, this should also prune out legitimate nodes which coincidentally acted in unison with the malicious subgraph.

The general attack strategy to detect a group of nodes that are the source of anomalous behavior in the network is done in three stages. In the first stage, an abnormality-aware graph neural network to first identifies nodes in the graph that are behaving abnormally. In this case, abnormal nodes are those which deviate from the behavior of its neighbors. From

there, the abnormality scores of each node are used as the new node embeddings and reverse all the arrows.

In the second stage, another graph neural network is applied, which propagates the abnormality score through the network in reverse by the same number of iterations as the forward direction. This way, the graph neural network should be able to assign to each node a score based on its contribution to the overall abnormality detected within the graph. Nodes with similar contributions are then grouped into single nodes. This can be done by grouping with staggered thresholds, much like how a multi-stage graph is constructed.

In the final stage, the method from stage one is used again, but this time with an additional embedding tracking where the cumulative contributions of each grouped node go. The grouped node with the highest contribution to the reconstruction of the abnormality score is then labeled as abnormal for each abnormal node identified in the first stage.

With the featurized data, GNNs can be trained to learn the underlying patterns of normal network activities. The graph structure allows the model to capture dependencies and correlations between different entities in the network. During the training process, the GNN learns to recognize the expected patterns of communication and relationships, establishing a baseline for normal behavior.

Once trained, the GNN can be applied to new network traffic data. Anomalies are identified by observing deviations from the learned patterns. The model can recognize unusual patterns of communication, unexpected connections, or abnormal traffic flows. The versatility of GNNs in capturing

complex relationships enables them to detect anomalies that may not be apparent through traditional methods.

The anomaly detection results obtained from the GNN can be integrated into network security alerting systems. When the model identifies suspicious behavior, alerts can be generated, notifying network administrators or automated response systems. This integration enhances the proactive nature of network security, allowing for timely responses to potential threats.

During the overview presented in the paper Graph Anomaly Detection With Graph Neural Networks: Current Challenges [1], the authors introduce the state of the GNN research field today, including some of the different types of problems that GNNs are applied to. In the static graph research area, the authors present three different types of problems: anomalous node detection, anomalous edge detection, and anomalous subgraph detection. It appears that the largest amount of work has been done in the node detection areas.

Looking at anomalous edges or subgraphs could be an opportunity for experimentation and contribution to the current literature. The authors note, "Anomalous subgraph detection is far more challenging than anomalous node or edge detection." [1] Since the authors claim that "Research on such anomalous edge detection in a static graph has been relatively limited." [1]

We are also presented with the idea of dynamic graph anomaly detection. The authors clarify the meaning of a static versus a dynamic graph by stating "Unlike a static graph, temporality is an important factor in a dynamic graph whose structure or attributes change over time." Since our problem of cybersecurity anomaly detection takes place in real time with a

changing graph, this topic is extremely relevant to the scope of our project in terms of real-world applicability. The authors highlight work in node and edge detection, but it seems that there is no current literature addressing dynamic anomalous subgraphs using GNNs. Depending on the difficulty of the problem, this could be another opportunity to add to the existing literature by constructing a GNN which detects dynamic subgraph anomalies or investigates the possibility of doing so.

In addition to this explanation of the current literature, the authors finish up by giving a few suggestions of problems they think are worth exploring, given the research landscape at the moment. It would be worthwhile to explore the possibility of incorporating some of these ideas when my team sits down to design our own GNN. The below figure summarizes the GNN methods mentioned in this overview paper.

This overview paper was the first thing my sponsor gave the team to read and we ended up with a lot of questions. While the paper delivered a broad view of how GNNs worked, it didn't dive into the technical specifics and discussed most of the algorithms it mentioned at a higher and more abstract level. The paper also lists many current or leading methods, but fails to mention or compare the performance of these methods with any type of results or statistics. Finally, the authors threw around a lot of acronyms and technical GNN terms without explanation, which was confusing to someone just learning about the topic for the first time.

At this point, my group began to dive into the more technical structure of what exactly a GNN was and how it worked. We accomplished this with the help of two interactive papers: A Gentle Introduction to Graph Neural Networks [2] and Understanding Convolutions on Graphs [9].

Essentially, the GNN pipeline is very similar to that of a Convolutional Neural Network. The difference is the convolutional versus graph blocks. A Convolutional Neural Network takes a grid as input and compute a convolutional operation multiple times (with some other operations such as pooling between layers).

The purpose of this is to extract features from the grid (typically a digital representation of an image) in order to learn on in a more traditional multi-layer perceptron neural network in the last layers of the network. Similarly, the Graph Neural Network takes a graph as input and then computes on that graph in order to achieve a transformed version of the graph which can then be trained on a more traditional neural network structure to make an anomaly detection or to tell us more about the graph. This process is illustrated in the figure below. [12]

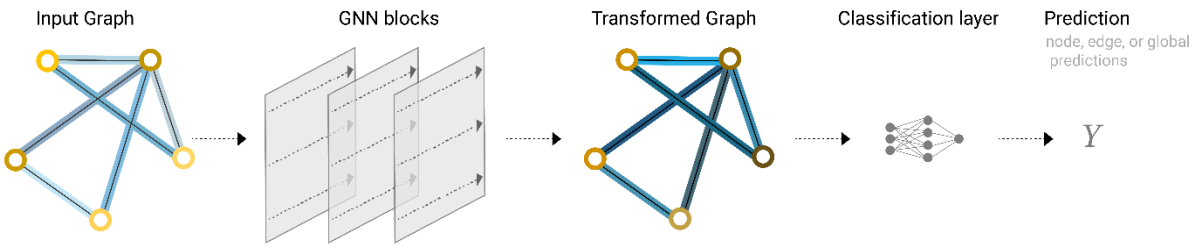


Figure 7. Illustration of the Graph Neural Network machine learning pipeline. [12]

The message-passing computation, which makes up the meat of the graph operation in the graph blocks can be summarized using the following equation [1]:

$$\begin{aligned}\mathbf{m}_u^{(k-1)} &\leftarrow \mathrm{AGGREGATE}^{(k-1)}(\{\mathbf{h}_v^{(k-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}),\\ \mathbf{h}_u^{(k)} &\leftarrow \mathrm{UPDATE}^{(k-1)}\left(\mathbf{h}_u^{(k-1)}, \mathbf{m}_u^{(k-1)}\right),\end{aligned}$$

[13]

Initially I, along with the majority of my team, found this equation confusing. However, the interactive part of the paper illustrates this operation fairly well. Figure 8 is one illustration of information passing through the graph layers of a GNN.

The nodes pass information to the nodes they are connected to, which gives us information about the connectivity of the graph and retains the context of the graph structure. The paper explains that this is especially important, since many other approaches may lose the context that the graph connectivity provides. The authors provide another graphic to illustrate this, where the message passing process pools the neighboring embeddings for a single node (these are called messages, hence the name message-passing) and then passes these through a neural network with a traditional network structure. [12]

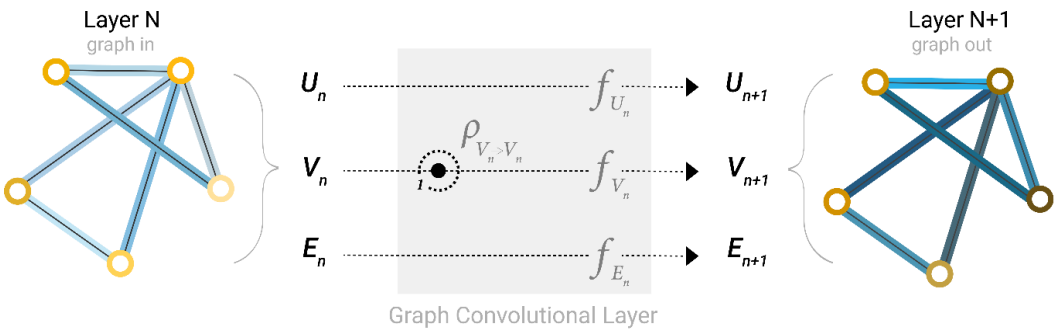


Figure 8. Message-passing in a Graph Neural Network. [12]

By reading these papers as a group, we began to round out our idea of what a Graph Neural Network was, how it worked, and the current state of the research field. Our sponsor has suggested another paper called Edge Detect: Edge-centric Network Intrusion Detection using Deep Neural Network [19] as well.

This paper purports to address the application of a Recurrent Neural Network approach to a cybersecurity problem on the UNSW15 dataset [1]. Reading this paper helped our team to understand how the application of the Graph Neural Network can increase prediction accuracy on a dataset that the team is already quickly becoming familiar with via experimentation.

In addition to this reading, the sponsor for my project has recommended a number of coding tutorials, beginning with a high-level introduction to neural networks, a Scikit-learn tutorial, and some examples and tutorials on PyTorch Geometric [20]. Geometric is a library which is specifically dedicated to the creation of Graph Neural Networks, and was extremely useful during the implementation of the GraphSAGE GNN.

The sponsor also recommended a book called “Graph Representation Learning” on Graph Neural Networks and graph representations in general [11]. We used this book to help us understand some of the more theoretical aspects related to the topic. While most of our work relied on other papers and sources, this book could be a good asset for future groups continuing this project.

The depth of insight offered by the book provides a solid foundation, enabling us to navigate the intricacies of GNN development with a more nuanced understanding. Armed with this targeted knowledge, the likelihood of falling into common pitfalls and rookie mistakes in the design and experimental phases decreases, setting the stage for a more robust and informed approach to our GNN implementation. This foresight underscores the potential instrumental role that this book may play in bolstering the theoretical underpinnings of our project, ultimately contributing to its overall success.

Anomalies are defined as “one that appears to deviate markedly from other members of the sample in which it occurs” [21]. [22]

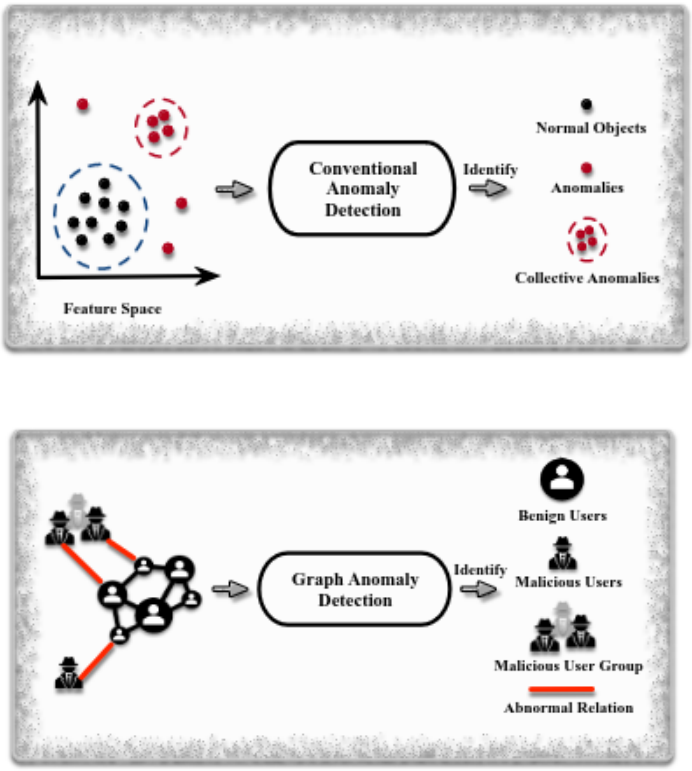


Figure 9. An illustration of conventional anomaly detection versus graph anomaly detection processes. [22]

Graph Anomaly Detection differs from conventional anomaly detection in that it doesn't just cluster anomalies based on a vector embedding, but through its relationship between other users. [22]

Surveys	AD	DAD	GAD	GADL				Source Code	Dataset	
				Node	Edge	Sub-graph	Graph		Real-world	Synthetic
Our Survey	●	●	●	●	●	●	●	●	●	●
Chandola *et al.* [18]	●	-	-	-	-	-	-	-	-	-
Boukerche *et al.* [45]	●	◐	-	-	-	-	-	-	-	-
Bulusu *et al.* [46]	●	●	-	-	-	-	-	-	-	-
Thudumu *et al.* [44]	●	○	○	-	-	-	-	-	-	-
Pang *et al.* [24]	●	●	○	○	○	-	-	●	●	-
Chalapathy and Chawla [47]	●	●	-	-	-	-	-	●	●	-
Akoglu *et al.* [25]	●	-	●	-	-	-	-	-	-	-
Ranshous *et al.* [48]	●	-	●	-	-	-	-	●	-	-
Jennifer and Kumar [49]	●	-	●	-	-	-	-	-	-	-
Eltanbouly *et al.* [50]	●	◐	○	-	-	-	-	-	-	-
Fernandes *et al.* [51]	●	◐	●	-	-	-	-	-	-	-
Kwon *et al.* [52]	●	◐	-	-	-	-	-	-	○	-
Gogoi *et al.* [53]	●	○	-	-	-	-	-	-	-	-
Savage *et al.* [54]	●	-	●	-	-	-	-	-	-	-
Yu *et al.* [6]	●	-	●	-	-	-	-	-	-	-
Hunkelmann *et al.* [3]	●	-	○	-	-	-	-	-	-	-
Pourhabibi *et al.* [19]	●	◐	●	◐	-	-	-	-	-	-

* AD: Anomaly Detection, DAD: Anomaly Detection with Deep Learning, GAD: Graph Anomaly Detection.
* GADL: Graph Anomaly Detection with Deep Learning.
* -: not included, ○ (1-2 references included), ◐ (3-10 references included), ● (10+ references included).

Figure 10. "A Comparison Between Existing Surveys on Anomaly Detection. We mark edge, sub-graph and graph detection as in our survey because we review more deep learning based works than any previous surveys." [22]

4.1.10 Dynamic Graphs

One of the biggest problems with dynamic graph neural networks is the computationally expenses to constantly update the network. Also, adding a new node means that there is no information of that node, making it hard to predict the normal and anomalous behavior of the node. Another major issue regarding dynamic graphs is that there are lots of subgraph changes because of the new nodes. Delving deeper into dynamic graphs, we examined structural temporal graph neural networks. Structural temporal graph neural networks are a type of subgraph anomaly detection.

One of the differences with anomalous detection on dynamic graphs is that you cannot find an anomalous edge or node on one timestamp, which

differs from static graphs. This problem occurs because the node and edge can change at any time. Structural Temporal graph neural networks solve this problem by looking at the structure over a short period of time and observing the changes. The dynamic graph is modified into a smaller, more meaningful subgraph that is then checked for abnormalities.

The previous models mentioned work on static graphs. However, there is also interest in developing tools for detecting anomalies in dynamic graphs. Since these are one of the more difficult problems to get optimal algorithms for, there is very little literature on the subject. The general attack strategy involves restricting to a sliding window where an autoencoding model attempts to learn evolving patterns.

A representative state-of-the-art model that uses this exact technique is the *dynamic evolving graph convolutional network* (DEGCN) [23]. It consists of three stages. The first stage uses a sliding window to generate multi-scale graphs. That is, graphs whose nodes represent a collection of nodes in the original graph connected by some random walk. It is multi-stage in the sense that the random walk can be seeded with multiple values and taken through multiple iterations to generate graphs that provide different subgraph representations of the original graph.

The second stage uses a graph autoencoder model to extract features of the node embeddings in the multi-scale graph which corresponds to patterns found along the time axis. Finally, The third stage combines the features extracted from the second step and feeds it through a standard neural network (multi-layer perceptron network) [24] to get an abnormality score at both the node-level and graph-level.

A Static Graph Network is a network whose nodes and edges don't change over time. Dynamic representations are the counter to this, their edges, nodes, and topologies do change over time. Dynamic representations can be good for PCAP data since requests become less significant over time due to the large number of connections being made in a short span of time and the limited time that a session exists for.

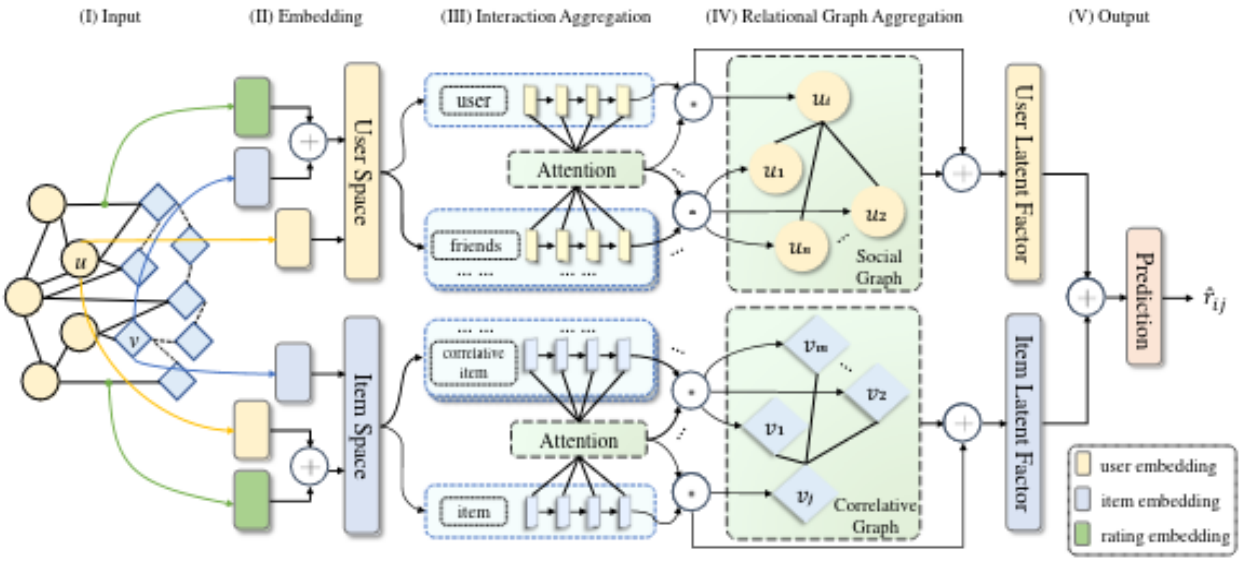


Figure 11. "Overview of GNN-DSR." [25]

The model converts the graph into User and Item embeddings. The user represents entities which in the context of PCAP data means routers or network endpoints, while items represent events or trafficking patterns. Items and Users are inputted into different embeddings. These embeddings

are then used to find short-term dynamic interests and long-term static interests.

4.1.10.1 Loss Function

$$= \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{R}} \|\mathbf{z}_u^\top \mathbf{R}_\tau \mathbf{z}_v - \mathcal{A}[u, \tau, v]\|^2, \quad [13]$$

The function takes every node connection and inputs each of the nodes and edges into the encoder independently, then inputs the result into a decoder. The result of the decoder is then compared to an adjacency tensor.

4.1.10.2 Short-Term Dynamic Interests

$$\mathbf{h}_i^S = \mathbf{LSTM}(\mathbf{X}(i)), \quad \mathbf{h}_j^S = \mathbf{LSTM}(\mathbf{Y}(j)), \quad [13]$$

The Long-Short Term Memory is used in this case because it works sequentially which allows it to be dynamic. The X(i) in this cause represents the user embeddings while the y(j) represents the item embeddings

4.1.10.3 Long-term Static Interests

$$\mathbf{h}_i^L = \sigma(\mathbf{W}_0^{uv} \sum_{j \in \mathcal{N}^{uv}} \alpha_{ij} \mathbf{x}_{i \leftarrow j} + \mathbf{b}_0^{uv}), \quad \mathbf{h}_j^L = \sigma(\mathbf{W}_0^{vu} \sum_{j \in \mathcal{N}^{vu}} \alpha_{ji} \mathbf{y}_{j \leftarrow i} + \mathbf{b}_0^{vu}), \quad (4)$$ [13]

This uses a Graph Attention Network, a modification of a normal multi-attention network to take into account edges. This is used to take into account information in the long term, but it is non-sequential.

4.1.10.4 Interactional Representation

The Short-term Dynamic Interests are combined with the Long-Term Static Interests to form the Interactional Representation through the equation:

$$\mathbf{h}_i^I = \mathbf{h}_i^S \odot \mathbf{h}_i^L, \quad \mathbf{h}_j^A = \mathbf{h}_j^S \odot \mathbf{h}_j^L.$$ [13]

The Interactional Representation is formed by the dot product of the Short-Term Dynamic Interests and the Long-Term Static Interests.

4.1.10.5 Relational Graph Aggregation

The User and Item interactional Representation is then used to create a Relational Graph Aggregation.

$$\mathbf{h}_i^N = f^{uu}\left(\mathbf{p}_i, \left\{\mathbf{h}_o^I : o \in \mathcal{N}_i^{uu}\right\}\right). \quad [13]$$

$$\mathbf{h}_j^N = f^{vv}\left(\mathbf{q}_j, \left\{\mathbf{h}_k^A : k \in \mathcal{N}_j^{vv}\right\}\right) \quad [13]$$

4.1.10.6 Latent Factor of User and Item

$$\mathbf{h}_i^u = g^{uu}\left(\left[\mathbf{h}_i^I, \mathbf{h}_i^N\right]\right), \quad \mathbf{h}_j^v = g^{vv}\left(\left[\mathbf{h}_j^A, \mathbf{h}_j^N\right]\right) \quad [13]$$

The Latent Factors are then aggregated in the output function

$$\hat{r}_{ij} = g_{\text{output}}\left(\left[\mathbf{h}_i^u, \mathbf{h}_j^v\right]\right), \quad [13]$$

4.1.11 Python Libraries

4.1.11.1 Pytorch

Besides getting a solid grip on Neural Networks, a significant chunk of our project's success heavily depended on the tools and frameworks we chose. With our focus squarely on the design and optimization of Graph Neural Networks (GNNs), we really couldn’t overlook the importance of selecting the right platforms. That's where PyTorch and PyTorch Geometric

(PyG) enter the picture. Having had the chance to look over their capabilities, our group was convinced these were the tools we needed, especially considering the complex nature of our project. Let's break down why.

Reading through the articles provided by our sponsor, we gained a strong understanding of PyTorch and its applications. In a general sense, it's a deep learning framework, and in recent times, it's been gaining traction in both the academic and industrial fields. What makes PyTorch stand out is how it is built - it's all about dynamic computational graphs. It meant that our team had the opportunity to play around with model development, making adjustments on the fly, without needing to dive deep into the code every single time.

Given our aim to develop a Graph Neural Network, PyTorch was the ideal platform choice. Its inherent design promotes quick experimentation, something we certainly took advantage of as we refined our GNN model.

When comparing with some other tools out there, we noticed PyTorch uses what's known as dynamic or "eager" computation. Instead of predefining everything and then running the code, operations in PyTorch are executed as they're called, almost like having a conversation. This kind of real-time feedback is crucial for troubleshooting.

GNNs aren’t simple. There are bound to be hiccups along the way. PyTorch, with its dynamic computation, made it much easier for us to understand and adapt our model when issues cropped up during development.

PyG is a library built upon PyTorch, but built especially for geometric deep learning. PyG has efficient ways to manage graph data, which was a huge time-saver for us during the implementation of our project. PyG was one of our main tools for GNN development. With so many built-in features, we were able to use it to have frameworks to work with rather than starting from scratch on a relatively unfamiliar topic, giving us more time to refine and adapt the approach we used.

Based on our research and experience, PyTorch Geometric isn’t just about ease-of-use when working with GNNs. It's also built for speed, particularly when dealing with extensive graph datasets, thanks to its optimized CUDA kernels. In the case of anomaly detection, which is the main goal of our project, we have to prioritize runtime because of the amount of data we are using. The faster we are able to process data, the quicker we can spot potential issues, PyG was the perfect library to work with for our application.

The more we delve into this field, the more we see how the community around the tool could benefit our team. PyTorch, together with PyG, has a vast and active community. This means there's a ton of resources out there, from tutorials to pre-trained models. When we got stuck, there was usually someone out there who faced the same issue. As we navigate through the use of GNNs for anomaly detection, having this community was invaluable. Sharing challenges, solutions, and insights can only speed up our project's progress.

One thing we’ve learned is that PyTorch doesn’t live in isolation. It can work with a ton of other tools. Whether we're talking data visualization, data processing, or even getting our solution out into the real world, PyTorch is

built to play nice with other systems. Anomaly detection with GNNs is a multifaceted task. We needed to see results and process various data. Hopefully, future groups may even tie our solution into bigger systems. With PyTorch, we didn’t have to worry too much about making all these different pieces fit together.

The deeper we dive into the world of Graph Neural Networks and anomaly detection, the clearer the picture becomes. It's not just about understanding the theory. The tools and platforms we opt for played a pivotal role. From what we can tell, vanilla PyTorch, coupled with PyTorch Geometric, offers us a solid foundation. With these in our toolkit, we were able to meet the demands of our sponsor while producing a solid implementation of a graph neural network. With this strong knowledge of tools like PyTorch, we began our project with a better understanding of what work we needed to put in to achieve our final goals.

Tensors:

Tensors are similar to a numpy array, but their operations can be done in the GPU. Due to the non-sequential nature of many of the operations needed for a neural network to train and make predictions, these operations can be parallelized and done at a larger scale using a GPU. Tensors can be scalers, arrays, or multi-dimensional arrays.

It can hold from 8 to 64 bit integer or float values. Tensors also allow for each neural layer to be back propagated globally through one optimization and loss function. This can be done because it keeps track of operations through a Directed Acyclic Graph, where the nodes are tensors and the directed edges are operations tensors do on other tensors. The

.backward() function goes backward through this graph to determine the gradients for each tensor. [26]

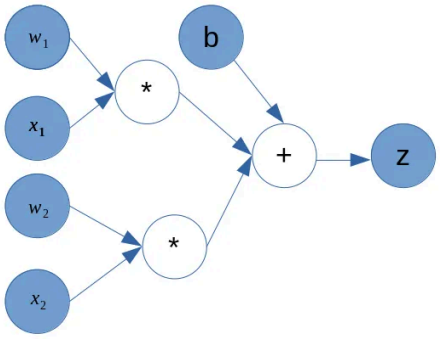


Figure 12. Example of a Compute Graph expression. [26]

The prediction of the model can be input into a loss function along with the actual value and because of the Directed Acyclic Graph, the model knows the gradient of each tensor with respect to the loss value. Running optimizer.step() will take these gradients and use them to update each tensor.

```
model = SimpleNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
inputs = torch.randn(100, 10)  # 100 samples, 10 features each
targets = torch.randint(0, 2, (100,))  # Random target labels
output = model(inputs[0])
loss = criterion(output)
optimizer.zero_grad() # clears gradients
loss.backward()
optimizer.step()
```

In this example we train a model on randomly generated data. We do this by instantiating a loss function, which determines how "off" the prediction is from the actual result. We also instantiate an optimizer function which uses the gradients from the calculation of the loss function using the output to change the tensors.

```
100 temp = torch.tensor([1,2,3])
```

This code converts an array into a tensor, this can also be applied to NumPy arrays. This is useful when needing to convert a dataframe that was imported from a csv file into a tensor, so that it can be inputted into a pytorch model.

```
102 temp = torch.ones((2,2))
```

This form of instantiating arrays can generate a tensor by the dimensions given in the parameter. These values don't necessarily have to be ones but can be zeros, random, or junk memory. This is useful for instantiating neural weights or random test input.

```
104 temp = torch.arange(0, 10 , 2)
```

torch.arrange generates values similar to the range function, but instead of a generator it creates a tensor. It creates a tensor with the values from the first parameter and exclusive to the second parameter with the third parameter being the step.

```
106 temp = torch.linspace(0, 1, 5)
```

torch.linspace generates values within a range such that they are evenly distributed. The first and second parameters are the boundaries and the third parameter is the amount of points within the boundary.

torch.nn.Linear:

This acts as a layer, taking in input of an array the size of the first parameter and multiplying it by the objects matrix, which is dimensioned such that its output is the size of the second parameter. This matrix is a trainable tensor in the model, meaning that backpropagation changes the weights of the matrix and possibly biases if biases are being used.

```
110 weight = torch.nn.Linear(2, 2, bias=True)
```

torch.nn.Embedding: [27]

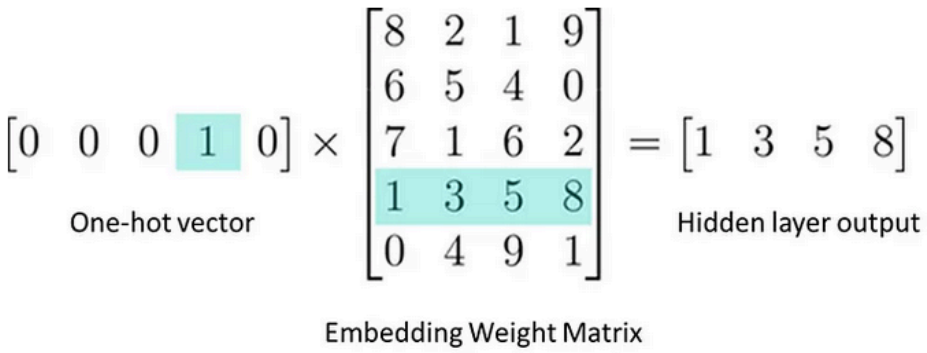


Figure 13. torch.nn.Embedding vector representation [27]

torch.nn.Embedding is used to give a vector representation to nominal data. Nominal data is just words in a fixed vocabulary. It works similarly to torch.nn.Linear but the vector that it takes as input is usually a one-hot vector meaning that its output is usually one of the rows. This is because it is used as a lookup table for what a word means in the context of the model through a numerical vector representation.

```
112 weight = nn.Embedding(vocab_size, vocab_dimensions)
```

Torch.nn.Parameter:

torch.nn.Parameter is similar to torch.nn.Linear in that it acts as a convolutional layer taking in an array of a given dimension and outputting an array of possibly another dimension, by multiplying the inputted array by its matrix. But torch.nn.Parameter is instantiated by a tensor, allowing it to have predefined weights, as opposed to torch.nn.Linear which uses random weights.

```
108 weight = torch.nn.Parameter((torch.FloatTensor(2,2)))
```

Models:

Pytorch models have at least two functions. The init function which is run when a model object is instantiated and the forward function which is called when a model object is given input.

```
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(10, 20)
        self.fc2 = nn.Linear(20, 2)
```

In the example code the init function is creating the layers for the model but it can also create other functions that can be used when a model is given input.

```
def forward(self, x):
    x = torch.relu(self.fc1(x))
    x = self.fc2(x)
    return x
```

This code handles model input using the functions declared in the init function.

Message Passing:

```
61 from torch_geometric.nn import MessagePassing
62
63 class CustomGNNLayer(MessagePassing):
64     def __init__(self):
65         super(CustomGNNLayer, self).__init__(aggr='mean')  # Mean aggregation
66
67     def forward(self, x, edge_index):
68         # x represents node features, edge_index represents graph connectivity
69         return self.propagate(edge_index, size=(x.size(0), x.size(0)), x=x)
70
71     def message(self, x_i, x_j):
72         # x_i: features of the central nodes
73         # x_j: features of the neighboring nodes
74         # Compute messages from node j to node i
75         return x_j
76
```

The message passing class takes in the features of the nodes and the edge between the nodes which is carried into an aggregate function along with connecting node edges. The aggregation function takes either the

mean, sum, or max of all messages being passed to the node. The value generated from the aggregation function is then added to the node embedding. The propagate function uses the edge_index to orchestrate the order of message, aggregation, and update function.

torch_geometric.utils.add_self_loops:

torch_geometric.utils.add_self_loops takes in the edge index and the number of nodes in the graph as first and second parameters respectively and returns another edges index such that each node is connected to itself

torch_geometric.utils.degree:

torch_geometric.utils.degree takes in the edge index as a parameter and returns a tensor for the degrees of each node. This is usually done for data normalization before propagation through message passing.

4.1.11.2 Numpy

NumPy is a Python library that provides tools for numerical operations. It is commonly used for mathematical computations in various scientific and engineering applications. NumPy is the fundamental package for scientific computing in Python, and it serves as the foundation for many other libraries and frameworks in the Python ecosystem. The array object provided by NumPy is a powerful data structure that enables efficient storage and manipulation of large datasets.

A good comparison to show the efficiency of arrays in NumPs to compare it to normal lists in python. A list in Python needs to store lots of information about the data before even looking at the data. It needs the size, reference count, object type, and object value. Also as the data is not stored consecutively and uses pointers, it wastes some computing time. NumPy improves this by storing all the data consecutively, and uses the minimal amount of space storing the data type. This causes it to be much faster as reading in less information into memory, parsing less information into memory and no need to type check.

Arrays in NumPy are one-dimensional or multidimensional, and they can hold elements of any data type. They can be created from Python lists or other iterable objects. One of the key features of NumPy arrays is their ability to perform element-wise operations, such as addition, subtraction, multiplication, and division. This vectorized approach to computation allows for efficient handling of large datasets, a crucial aspect in numerical computing and machine learning.

The ndarray, or N-dimensional array, is the primary data structure in NumPy. It is a flexible container for homogeneous data that can be manipulated using various array operations. NumPy arrays support broadcasting, which enables operations on arrays of different shapes and sizes. Broadcasting is particularly useful when working with datasets of varying dimensions, facilitating concise and expressive code. In addition to arrays, NumPy provides a range of mathematical functions that operate element-wise on arrays, making it a versatile library for scientific computing.

NumPy's array operations and mathematical functions are built on efficient, low-level implementations in C and Fortran, allowing for efficient

numerical computations. It integrates very well with other libraries, such as SciPy and Matplotlib. The ability to create and manipulate arrays efficiently makes NumPy an essential tool for tasks ranging from simple numerical calculations to complex scientific simulations.

4.1.11.3 Sci-kit Learn

DecisionTree:

Decision Trees are a classification model. They use classified data to classify more data. The Decision Tree creates a tree of feature thresholds that it inputs data from the top down to make a classification. Feature thresholds are conditionals that compare a feature from the data input with some predefined value. This predefined value is based on how many classifications that it separates, which is done through the Gini impurity function or the entropy function. This is done for every feature, and is ordered into a tree by how pure the feature threshold is. [28]

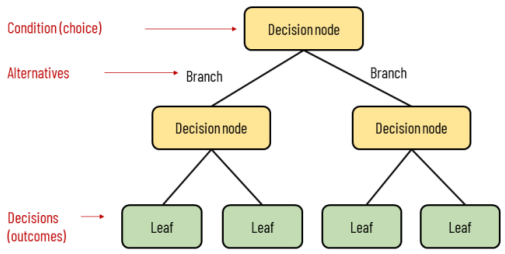


Figure 14. "Elements of a decision tree" [28]

The Gini impurity value subtracts 1 by the sum of squares of each probability that it will get a classification or the ratio that it produces a result when data is fitted to it. This makes it so the lowest possible value is when the feature threshold only produces one output making a result of 0 and it makes it so the highest possible value is when the feature threshold produces an equal distribution of outputs.

$$Gini = 1 - \sum_{i=1}^{C} (p_i)^2 \quad [30]$$

The Entropy impurity instead of squaring the probability of an output and taking the sum of all squares to subtract 1. It takes the negative of the probability times log base two. Since the probability is a fraction the negative is canceled out. The range of this function is greater but the most pure value is still zero while the least pure value is still the most distant value from zero.

$$Entropy = \sum -p_j log_2 p_j \quad [31]$$

Establishing a feature threshold is easily done when the feature is a Boolean or a nominal typed value. But when it is an Integer or a float value there can be an infinite number of feature thresholds. The decision tree handles this by ordering all the values for that feature in the data set, and

testing every value between two values for its Gini impurity and selecting the lowest value.

```
def find_best_split(self, X, y):
    m, n = X.shape
    if m <= 1:
        return None

    num_features = n
    best_gini = 1.0
    best_split = None

    for feature in range(num_features):
        feature_values = set(X[:, feature])

        for value in feature_values:
            left_indices = X[:, feature] <= value
            right_indices = ~left_indices

            left_gini = self.calculate_gini(y[left_indices])
            right_gini = self.calculate_gini(y[right_indices])

            weighted_gini = (len(y[left_indices]) / m) * left_gini + (len(y[right_indices]) / m) * right_gini

            if weighted_gini < best_gini:
                best_gini = weighted_gini
                best_split = ({"X": X[left_indices], "y": y[left_indices]},
                              {"X": X[right_indices], "y": y[right_indices]},
                              feature, value)

    return best_split
```

This function finds the feature threshold for each corresponding feature and orders it into a tree using the Gini impurity function.

```
def predict_example(tree, example):
    if 'class' in tree:
        return tree['class']
    if example[tree['feature']] <= tree['value']:
        return predict_example(tree['left'], example)
    else:
        return predict_example(tree['right'], example)

y_pred = [predict_example(model, example) for example in X_test]
```

The dictionary generated from the previous function can then be inputted into the predict example function with the second parameter being the input into the tree. The predict_example function checks if it has reached a leaf in the tree, if it does then it returns the leaf. If it hasn't reached a leaf, it checks which side of the feature threshold that the input falls under and it returns a recursive call to a node down in that direction.

RandomForest:

Random Forest is a classification model that uses a number of randomly generated decision trees. Each randomly generated decision tree is based off of a bootstrapped dataset: a dataset that is generated by randomly selecting values from another dataset that is inclusive of duplicate values. Each feature in each layer is selected from the best of two random non-repeated columns from the corresponding bootstrapped dataset.

The model makes predictions on new data by inputting the new data into each tree and returning the most common output. The model then uses the values not used in each decision tree's bootstrapped data set to test its accuracy. It then uses this measure of accuracy to decide how many variables in each step it randomly selects.

```
def find_split_point(X_bootstrap, y_bootstrap, max_features):
    feature_ls = list()
    num_features = len(X_bootstrap[0])
    while len(feature_ls) <= max_features:
        feature_idx = random.sample(range(num_features), 1)
        if feature_idx not in feature_ls:
            feature_ls.extend(feature_idx)
        best_info_gain = -999
        node = None
        for feature_idx in feature_ls:
        for split_point in X_bootstrap[:,feature_idx]:
            left_child = {'X_bootstrap': [], 'y_bootstrap': []}
            right_child = {'X_bootstrap': [], 'y_bootstrap': []}
            if type(split_point) in [int, float]:
                for i, value in enumerate(X_bootstrap[:,feature_idx]):
                    if value <= split_point:
                        left_child['X_bootstrap'].append(X_bootstrap[i])
                        left_child['y_bootstrap'].append(y_bootstrap[i])
                    else:
                        right_child['X_bootstrap'].append(X_bootstrap[i])
                        right_child['y_bootstrap'].append(y_bootstrap[i])
            else:
                for i, value in enumerate(X_bootstrap[:,feature_idx]):
                    if value == split_point:
                        left_child['X_bootstrap'].append(X_bootstrap[i])
                        left_child['y_bootstrap'].append(y_bootstrap[i])
                    else:
                        right_child['X_bootstrap'].append(X_bootstrap[i])
                        right_child['y_bootstrap'].append(y_bootstrap[i])
            split_info_gain = information_gain(left_child['y_bootstrap'], right_child['y_bootstrap
            if split_info_gain > best_info_gain:
                best_info_gain = split_info_gain
                left_child['X_bootstrap'] = np.array(left_child['X_bootstrap'])
                right_child['X_bootstrap'] = np.array(right_child['X_bootstrap'])
                node = {'information_gain': split_info_gain,
                        'left_child': left_child,
                        'right_child': right_child,
                        'split_point': split_point,
                        'feature_idx': feature_idx}
    return node
```
[32]

The code isn't exactly from the sklearn library. This code is from a user called Carbonati in a repository called random-forests-from-scratch but it functions similarly to how the sklearn library functions. The function generates a node in a decision tree for a random forest using pre-bootstrapped data and max number of features it can look at one time as its parameters.

It loops through until it looks at the maximum number of features it can look at once, in each loop it picks a random feature that hasn't already been looked at and it finds the point in-between each column of features where the most amount of information is gained, the information gained

function is nearly the same as the purity function in the Decision Tree but it uses entropy instead of Gini in the example. Once it finds that point it sets the current node to a dictionary containing the amount of information gained, the point in the array, and everything to the left and right of that point.

```
def split_node(node, max_features, min_samples_split, max_depth, depth):
    left_child = node['left_child']
    right_child = node['right_child']

    del(node['left_child'])
    del(node['right_child'])

    if len(left_child['y_bootstrap']) == 0 or len(right_child['y_bootstrap']) == 0:
        empty_child = {'y_bootstrap': left_child['y_bootstrap'] + right_child['y_bootstrap']}
        node['left_split'] = terminal_node(empty_child)
        node['right_split'] = terminal_node(empty_child)
        return

    if depth >= max_depth:
        node['left_split'] = terminal_node(left_child)
        node['right_split'] = terminal_node(right_child)
        return node

    if len(left_child['X_bootstrap']) <= min_samples_split:
        node['left_split'] = node['right_split'] = terminal_node(left_child)
    else:
        node['left_split'] = find_split_point(left_child['X_bootstrap'], left_child['y_bootstrap'], max_features)
        split_node(node['left_split'], max_depth, min_samples_split, max_depth, depth + 1)
    if len(right_child['X_bootstrap']) <= min_samples_split:
        node['right_split'] = node['left_split'] = terminal_node(right_child)
    else:
        node['right_split'] = find_split_point(right_child['X_bootstrap'], right_child['y_bootstrap'], max_features)
        split_node(node['right_split'], max_features, min_samples_split, max_depth, depth + 1)
```
[32]

This function uses recursion to define a Decision Tree in Random Forest. The bootstrap data in the left and right child is the information on the respective side of the split point. If either is empty or there is less than the predefined limit it turns into a terminal node, meaning that it returns the highest classification left. Otherwise it generates a node using the find_split_point function and it recursively calls itself, but the parent node is the node it is currently looking at and the depth is increased by one.

4.1.11.4 Pandas

Pandas is a Python library that provides tools for data analysis. It is commonly used for data preprocessing in machine learning tasks. The high performance combined with relatively simple interfaces makes it a very popular library. Pandas is built on top of the NumPy library, allowing for seamless integration between the two when manipulating data. The two types of classes that Pandas provides users are Series and DataFrames.

Series are one-dimensional objects capable of holding any data type. It can be created from a list, NumPy array, or a Python dictionary (key-value store). One of the key features of the Series data structure is its ability to handle missing values. Series also allows for vectorized operations such as addition, subtraction, multiplication, and division. This is especially useful in the field of machine learning where data is transformed into vectors with embeddings. Leveraging mathematical operations on this data allows developers to perform something like a nearest-neighbor search with cosine similarity.

The DataFrame structure is what Pandas is most known for. It is a two-dimensional object that follows a tabular structure. By using rows and columns, developers can easily manipulate their data during preprocessing. DataFrames can be created from many files in a very simple manner. Pandas provides developers with many IO tools to easily get data to and from a data frame. An example of this is the read_csv function that will turn a CSV file at some provided path into a Pandas DataFrame. Alongside this, Pandas allows DataFrames to be created from Excel, JSON, and HTML files, as well as SQL databases. Both data types support indexing which is especially useful when we want to split data into different sections such as a training and testing

set. During training the indexing can be used for batching, passing data in in batches can lead to more efficient training.

Dataframes can be instantiated through a dictionary where the key is the column name and the value is an array containing the values in the column from top to bottom.

```
data = {'Column1': [1, 2, 3],
        'Column2': ['A', 'B', 'C']}
```

Dataframes can also be instantiated with an array of dictionaries, where each dictionary represents a row in the dataframe and each key in each data frame represents a column with the value in the corresponding key the value in the column.

```
data = [{'Column1': 1, 'Column2': 'A'},
        {'Column1': 2, 'Column2': 'B'},
        {'Column1': 3, 'Column2': 'C'}]
df = pd.DataFrame(data)
```

They can also be instantiated with just arrays. A two dimensional array where each array represents a row in the dataframe and another array in the column parameter which represents the column name for each of the values in each of the arrays in the two dimensional array.

```
data = np.array([[1, 'A'], [2, 'B'], [3, 'C']])
df = pd.DataFrame(data, columns=['Column1', 'Column2'])
```

Data frames are also able to be imported from excel and SQL files.

```
df = pd.read_csv('your_file.csv')
```

Iloc uses Integers to index values in the data frame. As opposed to putting in the name of the column followed by the row that you need to access. Iloc uses an Integer, this allows the data frame to be treated as if it is a two dimensional array as opposed to something close to a dictionary.

4.1.12 Packet Capture

The way that we are able to know about the data communication between the machines is by using PCAP (packet capture) files. These files show us the raw data that is being passed through the network. PCAP files work by sniffing the network, which our graph neural network can analyze. We have used PyShark to be able to obtain live data for our program.

To get started on research into anomaly detection using graph neural networks, we first read about a paper on all the types of graph neural networks that exist and what are possibilities that we can consider. We are given PCAP files with the UNSW database, but it is much easier to modify the

data when it is in a csv format. We used PyShark to do this conversion so that this works for our live data that we receive in our simulation environment.

Packet capture files, typically in PCAP or pcapng format, serve as a record of network traffic, capturing individual packets exchanged between devices on a network. In the context of anomaly detection, these files become instrumental in understanding normal network behavior and identifying deviations that may indicate security threats or operational issues.

Anomaly detection using packet capture files faces challenges such as the sheer volume of data, diversity in network activities, and the need for efficient feature extraction. However, the richness of information in PCAP files presents opportunities for fine-tuned anomaly detection algorithms.

One of our tasks given by our sponsor was to create a script to featurize Packet capture files using PyShark which is a library for Python. Packet capture files, commonly in PCAP format, provide a granular view of network traffic, making them a valuable resource for anomaly detection systems. It was important for the concept of our project to outline the methodology and implementation of utilizing PyShark, a Python wrapper for Wireshark, to process PCAP files and extract features relevant to anomaly detection.

4.1.12.1 PCAP Parser

As previously mentioned, PyShark is a Python library that serves as a wrapper for Wireshark's packet dissection capabilities. It provides a

convenient interface for accessing information within PCAP files, enabling the extraction of packet-level details.

The provided Python script demonstrates the use of PyShark to featurize PCAP files. Features extracted include standard network attributes such as source and destination IP addresses, port numbers, packet length, and timing characteristics. Additionally, the script captures protocol-specific details like TCP flags.

The features extracted from the packet capture files using PyShark provide a comprehensive view of the network dynamics. These features, when organized into a graph structure, can represent the interactions and relationships among different entities within the network. Nodes in the graph can represent devices or endpoints, and edges can represent communication links between them.

Future research in this domain could explore enhancements to the featurization process, considering additional features or refining existing ones for improved anomaly detection performance. Furthermore, investigating the interpretability of the GNN model outputs could provide valuable insights into the specific characteristics that lead to anomaly classifications.

We were able to use the research we made to create a python script that took the wireshark pcap data and converted it to featurized data in a CSV.

imestamp	SourceIP	DestinationIP	SourcePort	DestinationPort	Protocol	Length
424219007.658518	175.45.176.3	149.171.126.16	22592	143	TCP	77
424219007.658559	175.45.176.3	149.171.126.16	22592	143	TCP	77
424219007.737404	149.171.126.16	175.45.176.3	143	22592	TCP	97
424219007.737414	149.171.126.16	175.45.176.3	143	22592	TCP	97
424219007.760103	175.45.176.0	149.171.126.16	62762	56430	TCP	64

4.1.13 GNN Frameworks

4.1.13.1 Graph Convolutional Networks (GCN)

Inspired by how standard convolutional neural networks can be thought of as learning on the spectral domain of an input signal, graph convolutional networks define a spectral domain analogue for graphs and hence a convolution analogue for graphs [33]. In particular, in a standard convolutional neural network, we can think of the input image as a two-dimensional wave signal. Applying a Fourier transform to the signal moves us from time domain to frequency domain which corresponds to the spectral domain of the associated Laplace operator on signals. Convolution in the time domain is then equivalent to multiplication in the frequency domain (read spectral domain).

For graphs, we define the Laplace operator on the adjacency matrix representation of a graph as the normalized difference between its degree and itself. This Laplace operator is independent of the specific adjacency matrix used to represent the graph. Since this operator is a linear transformation, there is a corresponding spectral domain for graphs.

The Fourier transform for graphs would then be the corresponding change of basis algorithm from vertex domain (as encoded by the adjacency

matrix) to spectral domain (corresponding to our Laplace operator). We now define convolution of graphs as the inverse fourier transform over matrix multiplication in the spectral domain. This convolution can be shown to be precisely an aggregator transformation that takes node data and propagates across node neighborhoods. [34]

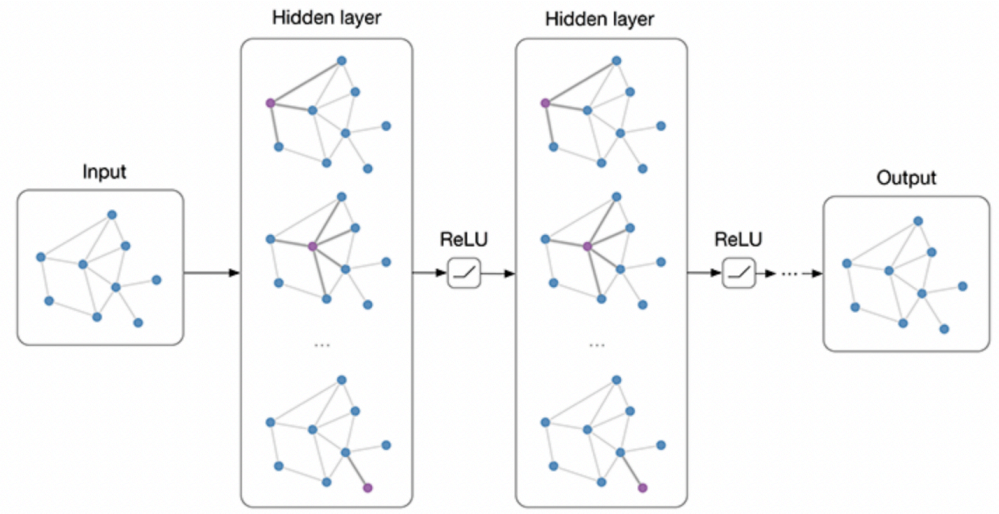


Figure 15. Illustration of a "Multi-layer Graph Convolutional Network (GCN) with first-order filters." [34]

An advantage to applying convolution in the spectral domain is that we can avoid node embedding aggregation and instead focus on fixed matrix multiplication. This allows us to leverage the full power of numerical linear algebra when performing computations and training using backpropagation. This also gives a precise unique decomposition of a graph into its spectral components which can be used to create an autoencoder for the graph that accurately extracts the necessary graph-level features.

A disadvantage to this approach is that the Laplace operator requires performing matrix factorization which is computationally expensive and does not scale well for larger graphs. To combat this, standard methods tend to restrict themselves to ordering the spectral data by eigenvalue and calculating the first couple of eigenfunctions to be able to apply a Fourier transform. This provides an approximation of the Laplace operator which may be sufficient for some graphs but not all.

4.1.13.2 GraphSAGE (Graph Sample and Aggregation)

As previously described, a GraphSAGE essentially attributes to each node of a graph some data called an *embedding* [35]. Parameterized sampling and aggregate rules then propagate and accumulate these embeddings between neighboring nodes to move to the next layer/step in the network. After some fixed number of iterations, the resultant embeddings correspond to the output of our GraphSAGE.

Training is done by learning the parameters via backpropagation as would a standard neural network. The result is a model that can learn node properties dependent on the local structure of the graph, i.e. the neighborhood structure. [36]

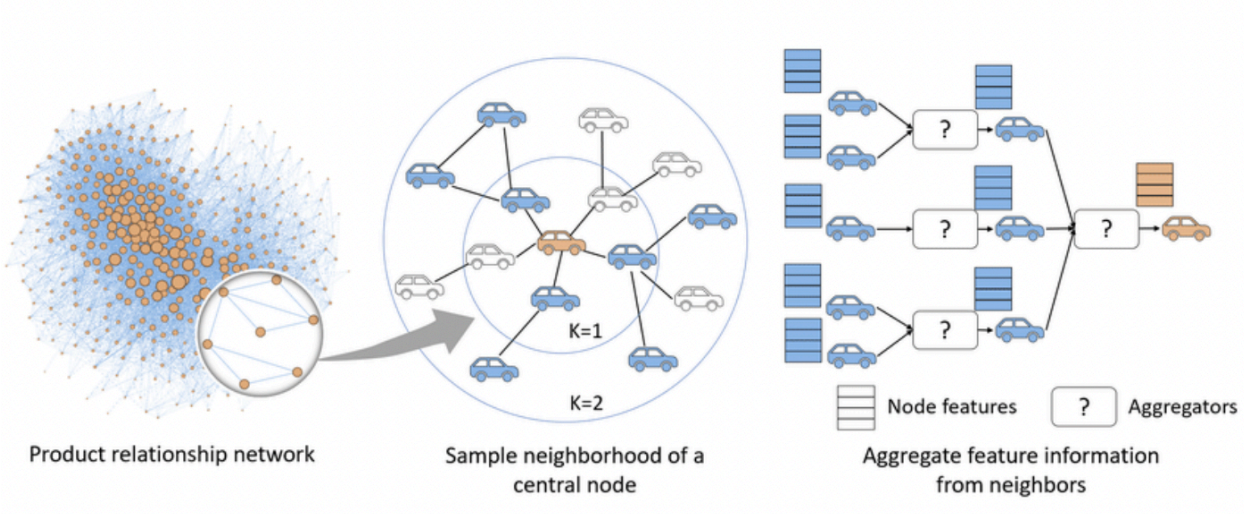


Figure 16. Illustration of the GraphSAGE feature aggregation. [36]

There are some major advantages to this approach. GraphSAGE is quick and easy to implement since it follows the same principle as a convolutional network but where adjacent pixels are now interpreted as neighboring nodes. This means that GraphSAGE effectively captures neighborhood information and can learn local graph structures efficiently. It moreover supports mean, LSTM, pooling, and other common convolution procedures.

However, there are also some disadvantages to this approach. GraphSAGE can only consider up to some fixed neighborhood depth when learning node properties which are defined as the iteration count. This means that trial and error would need to be done to identify the appropriate iteration count for learning some ideal property. The drawback is that properties dependent on global network structure cannot be learned via this

approach for arbitrary graphs since this requires increasing the iteration count proportional to the maximum path length of the graph.

4.1.13.3 Graph Attention Networks (GAT):

Graph Attention Networks (GATs), a recent advancement in the field of neural networks, offer an intriguing approach to dealing with graph-structured data. Originating from graph neural networks (GNNs), GATs incorporate attention mechanisms, a concept initially used in sequence-based models like those in language translation, to enhance their processing of graph data.

This integration of attention mechanisms allows GATs to allocate varying levels of importance to different nodes in a graph, which is essential for making more accurate predictions and gaining deeper insights. By doing so, GATs can focus on the most relevant parts of the graph, a feature that distinguishes them from traditional GNNs. [37]

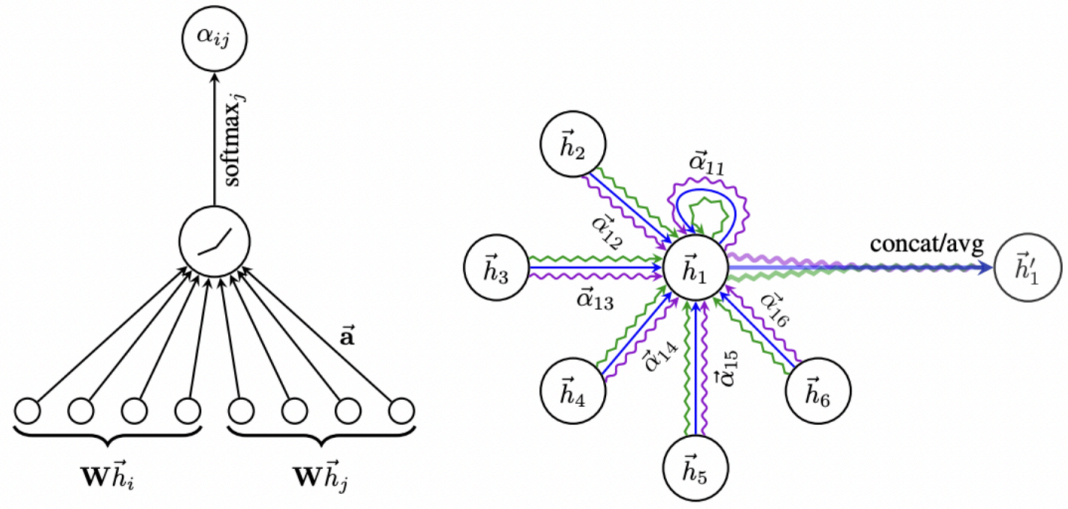


Figure 17. Illustration of Graph Attention Networks. [37]

GATs stand out because of their unique ability to adaptively learn which nodes in a graph are more significant, thereby focusing on these nodes during the learning process. This adaptability makes them suitable for various types of graph data, ranging from social networks to biological systems.

The attention mechanism employed in GATs ensures that they can selectively aggregate information from neighboring nodes. This selective aggregation leads to more precise learning outcomes, as GATs concentrate on the most relevant information available within the graph. Furthermore, GATs are capable of handling graphs with a diverse range of node types and connections, showcasing their versatility.

However, despite these strengths, GATs also come with certain limitations. One major drawback is their high computational demand, particularly when dealing with large graphs. The detailed attention calculations required by GATs can be resource-intensive, which poses a challenge, especially in scenarios involving big data. Additionally, fine-tuning GATs can be complex, as their intricate nature makes optimizing their settings a challenging task. This complexity can also result in difficulties when scaling GATs for very large graphs, as their efficiency tends to decrease with increasing graph sizes. Another potential issue with GATs is their tendency to overfocus on certain graph areas, which can lead to overlooking broader patterns or relationships within the graph.

In the context of anomaly detection, GATs offer a promising approach. Anomaly detection involves identifying unusual or unexpected patterns in data, and GATs are particularly adept at this task due to their ability to zoom in on the most relevant parts of a graph. This capability is invaluable in areas such as financial fraud detection or network security, where quickly identifying deviations from normal patterns can be crucial. By concentrating on key areas of a graph, GATs can effectively spot anomalies, making them a valuable tool in various sectors.

In conclusion, Graph Attention Networks represent a significant step forward in processing graph-structured data. Their precision and adaptability to different data types are some of their key strengths. However, they are not without challenges, as they require substantial computational power and can struggle with large graph sizes. Despite these challenges, the potential applications of GATs, particularly in anomaly detection, are vast and promising. As research in this area continues to evolve, we can expect GATs to become even more effective and versatile, enhancing their applicability across various domains that rely on graph data analysis.

4.1.13.4 ChebNet (Spectral-based Graph Convolutional Network):

ChebNet, [38] a spectral-based Graph Convolutional Network, marks a notable development in the realm of neural networks, particularly for handling graph-structured data. This network, stemming from the broader category of Graph Convolutional Networks (GCNs), utilizes spectral graph theory to process data in graph form.

ChebNet's approach involves leveraging Chebyshev polynomials to approximate graph Laplacians, enabling the network to capture the essence of graph structures effectively. This method offers a unique way of interpreting and analyzing data represented as graphs, making ChebNet stand out in the field of graph-based neural networks.

One of the primary strengths of ChebNet is its ability to efficiently encode graph topology into its learning process. By using Chebyshev polynomials, ChebNet can effectively approximate graph Laplacians, allowing it to capture the global structure of the graph. This capability makes ChebNet highly effective in understanding complex graph patterns, which is crucial for tasks involving large and intricate graph structures. Furthermore, ChebNet's spectral approach allows for a more comprehensive analysis of graph data, as it considers the overall structure of the graph rather than just the local neighborhoods.

Despite these advantages, ChebNet also faces some challenges. One significant limitation is its dependency on the fixed graph structure. This reliance means that ChebNet struggles with dynamic graphs where the structure changes over time. Additionally, the spectral approach used by ChebNet can be computationally intensive, particularly when dealing with large-scale graphs. This computational demand can make ChebNet less practical for applications involving very large or rapidly evolving graph data. Moreover, ChebNet's reliance on the entire graph's spectral properties can limit its ability to generalize to graphs with different structures.

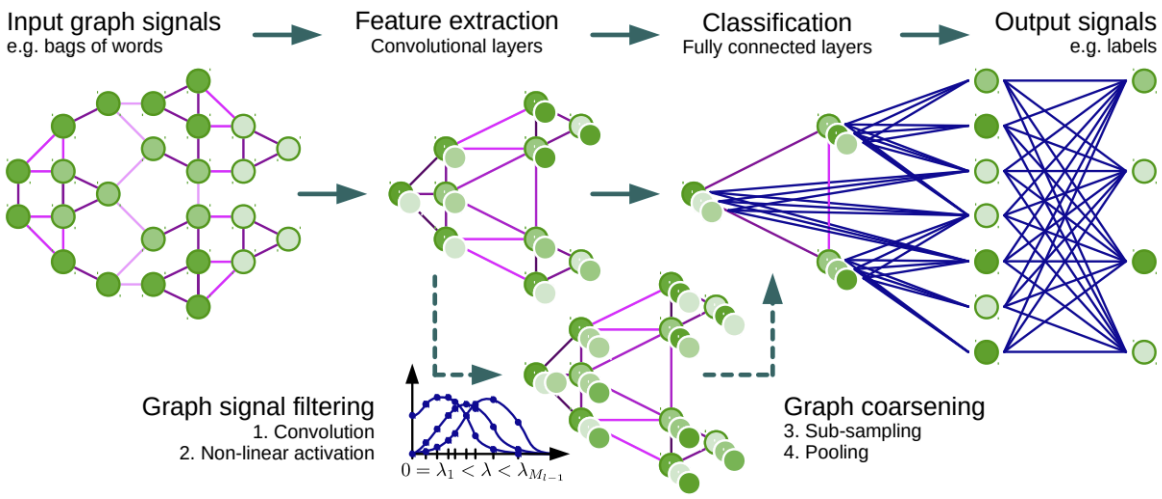


Figure 18. Diagram of CNN graph architecture as presented in the ChebNet paper. [38]

In the area of anomaly detection, ChebNet offers a promising solution. Anomaly detection, which involves identifying patterns in data that deviate from the norm, can benefit greatly from ChebNet's ability to analyze the overall structure of graphs. This global perspective enables ChebNet to identify unusual patterns or outliers in graph data that might not be evident when focusing solely on local neighborhoods. This capability is particularly useful in scenarios like network security or fraud detection, where understanding the entire graph's structure is crucial for spotting anomalies.

In summary, ChebNet represents a significant advancement in graph-based neural networks, particularly due to its spectral approach to graph convolution. Its strength lies in its ability to capture and analyze the global structure of graphs, making it effective for tasks that require a comprehensive understanding of graph data. However, its reliance on a fixed

graph structure and the computational intensity of the spectral approach pose challenges, especially for dynamic or very large graphs. Despite these limitations, ChebNet's potential in applications like anomaly detection is considerable, offering a new perspective in identifying unusual patterns in complex graph data. As research in this field continues, further enhancements in ChebNet's design and functionality can be expected, broadening its applicability and effectiveness in various graph-based data analysis tasks.

4.1.13.5 Graph Isomorphism Network (GIN):

Graph Isomorphism Network (GIN), [39] a distinctive addition to the landscape of graph neural networks, has garnered attention for its novel approach to graph-structured data. As an evolution within the graph neural network paradigm, GIN is designed to address a fundamental question: how to effectively determine if two graphs are isomorphic, that is, structurally identical. The GIN model achieves this by employing a unique architecture that enables it to capture the intricate subtleties of graph structures more effectively than traditional graph neural networks. [39]

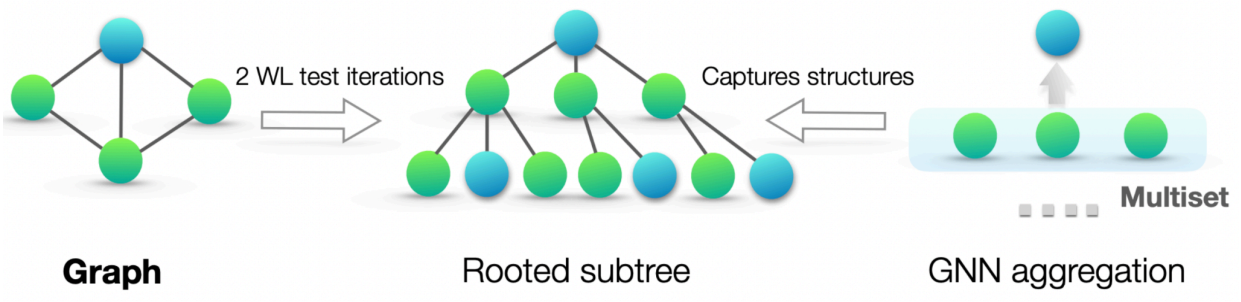


Figure 19. GIN structure outline. [39]

One of the key strengths of GIN lies in its exceptional ability to discern fine-grained structural differences between graphs. This capability stems from its design, which closely aligns with the Weisfeiler-Lehman (WL) test of graph isomorphism. By iteratively updating node representations in a manner that preserves the graph's structural information, GIN can effectively identify nuanced patterns and relationships within the graph. This makes it particularly adept at tasks that require a deep understanding of graph topology, such as chemical compound analysis or social network studies.

However, GIN also encounters certain challenges. Its sensitivity to structural nuances, while a strength, can also be a drawback. In scenarios where the graph data is noisy or where small structural variations are not significant, GIN's focus on fine details might lead to overfitting. Additionally, like other graph neural networks, GIN can face scalability issues when dealing with very large graphs, as the computational complexity increases with the size and complexity of the graph.

In the context of anomaly detection, GIN offers intriguing possibilities. Anomaly detection involves identifying unusual or unexpected patterns in data, and GIN's ability to detect subtle structural differences makes it well-suited for this task. For instance, in network security, GIN can be used to identify unusual patterns in network traffic that might indicate a security breach. Similarly, in financial fraud detection, GIN's capability to spot atypical transaction patterns in financial networks can be invaluable.

In summary, Graph Isomorphism Network represents a significant step forward in the field of graph neural networks. Its ability to closely mimic the Weisfeiler-Lehman test for graph isomorphism enables it to capture detailed structural information in graphs, making it highly effective for tasks requiring intricate graph analysis. While it faces challenges such as potential overfitting and scalability issues, its capabilities in applications like anomaly detection are noteworthy. As research in this domain continues to evolve, we can expect further advancements in GIN, enhancing its effectiveness and broadening its range of applications in graph-based data analysis.

4.1.13.6 Message Passing Neural Network (MPNN):

Message Passing Neural Network (MPNN), [40] a significant model in the realm of graph neural networks, has emerged as a powerful tool for dealing with graph-structured data. MPNN stands out due to its unique approach that centers around the concept of message passing. In this framework, nodes in a graph send and receive messages to and from their neighbors, and these messages are then used to update the nodes' representations. This process, repeated over several iterations, allows the

network to learn rich and complex representations of the graph, capturing both local and global structural information. [40]

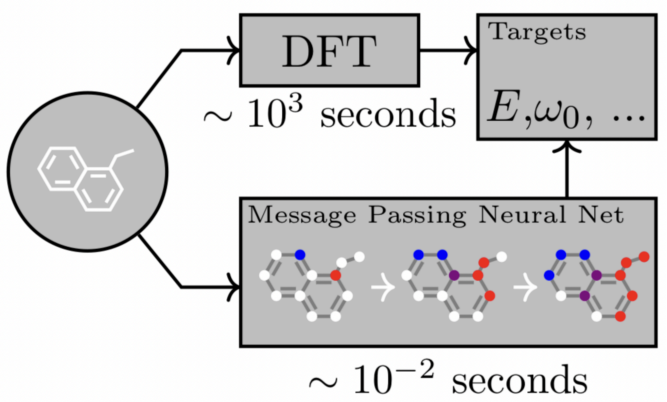


Figure 20. Message Passing Neural Network experiment overview with prediction on the "quantum properties of an organic molecule". [40]

A major advantage of MPNNs is their flexibility in capturing various graph properties. By allowing nodes to exchange information, MPNNs can adaptively learn the features of each node based on its neighborhood, leading to a more nuanced understanding of the graph structure. This ability makes MPNNs particularly effective in tasks that require an understanding of the relationships and interactions within a graph, such as molecular property prediction or social network analysis. Additionally, the message-passing framework can be tailored with different aggregation and update functions, providing versatility in how the network processes the graph data.

However, MPNNs also have some limitations. One of the primary challenges is their reliance on the local neighborhood structure. While this

focus on local information allows for detailed node-level analysis, it can sometimes limit the network's ability to capture broader graph-wide patterns. Moreover, MPNNs can struggle with scalability issues, particularly when dealing with large graphs, as the message-passing process can become computationally intensive. Additionally, there is a risk of over smoothing, where the node representations become too similar, losing valuable information about the graph's structure.

In the field of anomaly detection, MPNNs offer promising capabilities. Anomaly detection involves identifying patterns in data that deviate from expected norms, and the local neighborhood focus of MPNNs can be highly effective in this regard. For instance, in detecting fraudulent activities in financial networks, MPNNs can analyze transaction patterns at the node level, identifying unusual behaviors that might indicate fraud. Similarly, in network security, MPNNs can be used to monitor traffic at individual nodes, spotting anomalies that could signify security threats.

In conclusion, Message Passing Neural Networks represent an innovative approach in graph-based data analysis, with their message-passing mechanism enabling detailed and adaptable learning from graph structures. Their strength lies in their ability to learn complex node representations based on neighborhood information, making them suitable for a wide range of applications. However, challenges such as scalability and potential over smoothing need to be addressed. Despite these limitations, the application of MPNNs in areas like anomaly detection is particularly promising, offering new ways to identify unusual patterns in complex graph data. As the field progresses, we can anticipate further enhancements to MPNNs, expanding their utility and efficiency in graph-based neural network tasks.

4.1.13.7 Optimal Function Choice

Finding the best message/update functions can be challenging. Each type of GNN has its strengths and weaknesses, and the best choice depends on the specific problem and data at hand. We would say Graph Attention Networks (GAT) would be the best choice from what we have seen so far because of its ability to weigh different nodes differently, which can be crucial for highlighting anomalies.

In the first phase of our project, we have gone through the steps of setting up technologies and software in order to prepare for more of a hands-on implementation of our ideas in the following semester of Senior Design 2. Technologies that were necessary to set up for use: Wireshark, Github repo with sponsor, Jira board, box for files, Pytorch libraries. Almost all software was set up and ready to go for starting to get tangible products that our sponsor requests of us.

As getting packets on a network is something that happens over time, we did some research on graph neural network anomaly detection in time series. The first thing that needed to be done is graph structured learning. This is the learning stage of the graph neural network. This is what learns the relationships between the nodes and encodes this relationship as the edge. For this, a directed graph is given, the graph will not be created by the neural net, but will be given by the researchers as this is a static graph.

The next thing is graph attention-based forecasting, where this predicts what should happen in each node and what the edges should be like. This is using a forecast, where the script will forecast what the node

should be doing at that time. A feature extractor is used to capture the relationship between nodes.

The last step is to see the graph deviation scoring. This is known by the fact that in the last step the neural net predicts the future, and if it deviates it will be marked as an anomaly. Graph edges can be used to show the connection between nodes, the greater the connection the larger the weight of the edge.

We are taking Propagation Code Analysis Program data and using that data to detect anomalies using Graph Neural Networks. There are different architectures of GNNs and depending on the architecture of our Graph Neural Networks will affect how we format our PCAP data as a graph.

One Graph Neural Network type architecture we can use is a Graph Convolutional Network. It uses a multilayer perceptron like a Convolutional Neural Network, meaning that it has layers of hidden states that it uses to generate an output from input. But its hidden states on each layer correspond to a node on the network and each node is represented by some value. Each hidden state is equal to the ReLU of the summation of each adjacent node in a function. This function is defined as the product of the corresponding hidden state in the last layer multiplied by a weight that is changed by backpropagation over the root of the product of the connected nodes. The issue with this Graph Neural Network is that it doesn't take into account the properties of the edges other than the nodes that it connects.

A similar Graph Neural Network that is similar to Graph Convolutional Networks in how it connects the different layers of the hidden states between their corresponding adjacent nodes, Message-passing neural networks are able to take into account edge properties. It has a hidden state

in each layer that corresponds to each of the nodes like in a Graph Convolutional Network.

It accumulates the sum of the adjacent nodes by passing the previous hidden layer of the current node and adjacent node along with the edge properties between the corresponding node into a messenger function. It then takes the sum returned from that messenger function and passes it into the vertex update function along with the corresponding hidden state on the previous layer. The messenger function acts as an accumulator and the vertex update function acts like an activation function.

Like a Convolutional Neural Network input is fed into these hidden states to generate a predicted value and if the model is training, this predicted value is compared to the actual value and backpropagation through the weights of the model to generate a value closer to the actual value.

Another more modern Graph Neural Network is a Graph Transform Network. This version, unlike the other two, is based on the principles of the Transformer Model, which is typically used for Natural Language Processing. instead of words it takes in nodes and optionally edges of graphs. Unlike Transform Models in Natural Language Processing, the Graph Transform Network in each of the attention heads compares each of the adjacent nodes with the node that it is looking at and pools them into a sum.

```
class GCN(nn.Module):
    def __init__(self, num_features, hidden_dim, num_classes):
        super(GCN, self).__init__()
        # Define a GCN layer with 'num_features' input features and 'hidden_dim' output features
        self.conv1 = GCNConv(num_features, hidden_dim)
        # Define another GCN layer that takes 'hidden_dim' inputs and outputs 'num_classes' features
        self.conv2 = GCNConv(hidden_dim, num_classes)

    def forward(self, data):
        # The forward method takes a data object with edge_index and x attributes
        x, edge_index = data['x'], data['edge_index']

        # Apply the first GCN layer and a ReLU activation function
        x = F.relu(self.conv1(x, edge_index))
        # Apply dropout for regularization
        x = F.dropout(x, training=self.training)
        # Apply the second GCN layer
        x = self.conv2(x, edge_index)

        return F.log_softmax(x, dim=1)  # Apply log_softmax for the output layer

# Example usage
num_features = 16  # Number of input features
hidden_dim = 32    # Dimension of hidden layer
num_classes = 3    # Number of classes for classification
num_nodes = 10

model = GCN(num_features=num_features, hidden_dim=hidden_dim, num_classes=num_classes)
edge_index = torch.tensor([[0,1,2,3,4],[1,2,3,4,5]],dtype=torch.long)
# Example graph data structure
data = {
    'x': torch.rand((num_nodes, num_features)),  # Node features
    'edge_index': edge_index,  # Edges of the graph
}

output = model(data)
print(output)
# Assuming 'data' is a graph data object from a dataset like PyTorch Geometric's datasets
# The data object must have 'x' as the node feature matrix and 'edge_index' as the edge list
out = model(data)
```

This is template code for the Graph Convolutional Network which is a step up from the random forest in the previous lines of code. The current issue with its implementation is putting the pcap data into the form of nodes and edges. The initial idea is to treat the ip addresses in the pcap data as nodes and requests between them as edges. But there is the question of how the data in the requests is going to be represented.The edges could be given the properties of a pcap request, but there could be many requests

between two nodes. The best option that we can see is to have some kind of meta data point to represent all the requests directionally between two nodes. An example of this could be the number of requests made or the average size of requests.

```
class GNNLayer(MessagePassing):
    def __init__(self, in_channels, out_channels):
        # Initialize the message passing mechanism with the 'add' aggregation.
        super(GNNLayer, self).__init__(aggr='add')
        # Initialize the layer transformation.
        self.lin = torch.nn.Linear(in_channels, out_channels)

    def forward(self, x, edge_index):
        # Start the message passing process.
        # First, add self-loops to the adjacency matrix.
        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))
        # Then, apply the linear transformation to node features.
        x = self.lin(x)
        # Finally, start propagating messages.
        return self.propagate(edge_index, size=(x.size(0), x.size(0)), x=x)

    def message(self, x_j):
        # The message function simply returns the node features.
        return x_j

    def update(self, aggr_out):
        # The update function takes in the output of aggregation and can compute further transformations.
        # Here, we simply return the aggregated messages.
        return aggr_out

class GNN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super(GNN, self).__init__()
        self.conv1 = GNNLayer(in_channels, hidden_channels)
        self.conv2 = GNNLayer(hidden_channels, out_channels)

    def forward(self, data):
        x, edge_index = data['x'], data['edge_index']

        x = self.conv1(x, edge_index)
        x = torch.relu(x)
        x = self.conv2(x, edge_index)

        return x

# Assuming you have a graph with 16-dimensional node features and you want to perform a binary classification
num_node_features = 16
hidden_channels = 32
num_classes = 2
num_nodes=10

model = GNN(num_node_features, hidden_channels, num_classes)
edge_index = torch.tensor([[0,1,2,3,4],[1,2,3,4,5]],dtype=torch.long)
# Example graph data structure
data = {
    'x': torch.rand((num_nodes, num_node_features)),  # Node features
    'edge_index': edge_index,  # Edges of the graph
}

output = model(data)
print(output)
```

This is a template for what the sponsor wants as the end product. It is a Graphic Neural Network that uses the pytorch geometric library. It takes in a 2 dimensional tensor with two rows that represents the edges, the first dimension represents where the edge starts and the corresponding position in the 2nd row is where the edge goes. It takes in another 2-dimensional tensor that represents the nodes. It is defined as the number of nodes by the number of features in each node. Each row in the tensor is a representation of each node.

4.1.14 Random Forest

The team decided to each implement the random forest model individually and compare our results. We decided to use the RandomForestClassifier function from the Scikit Learn library in Python to train the model. Preliminarily, we achieved an accuracy of approximately 82% using all available features by splitting the data in our training file. It was possible to split the data this way since all of our data is labeled. Our results for each cyberattack category are shown below in the confusion matrix.

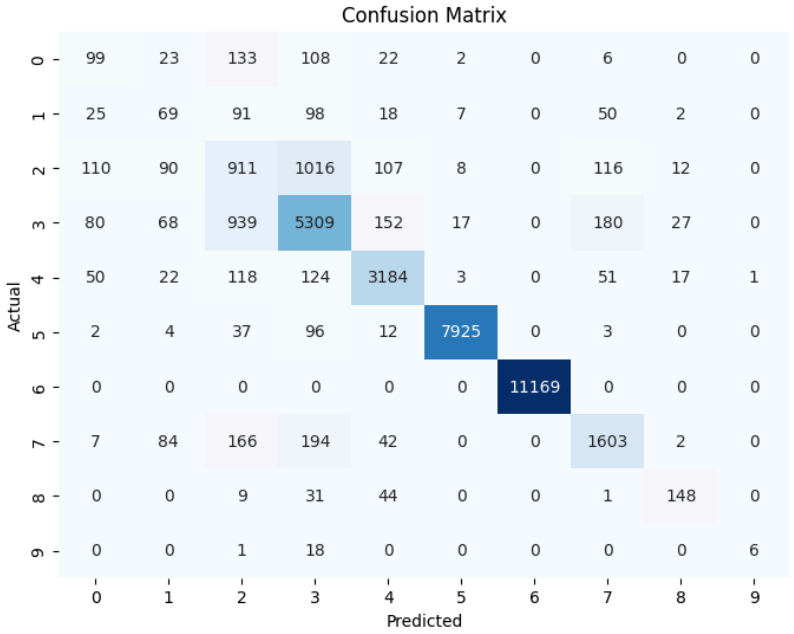


Figure 21. Preliminary results on training dataset split.

After testing on the entire dataset (files 1-4 of the UNSW dataset), we achieved an accuracy of 99.56 %. This dataset included 2,540,047 samples and we used an 80/20 split for training and testing respectively. Our final results for this data are shown in the figure below. We removed the features srcip, sport, dstip, dsport, and st_ftp_cmd before training after our mentor suggested removing the ip addresses and ports in order to reduce overfitting on irrelevant features. The st_ftp_cmd variable was removed due to a mixed data type issue that we hope to resolve as we continue using the random forest models to perform feature importance analysis.

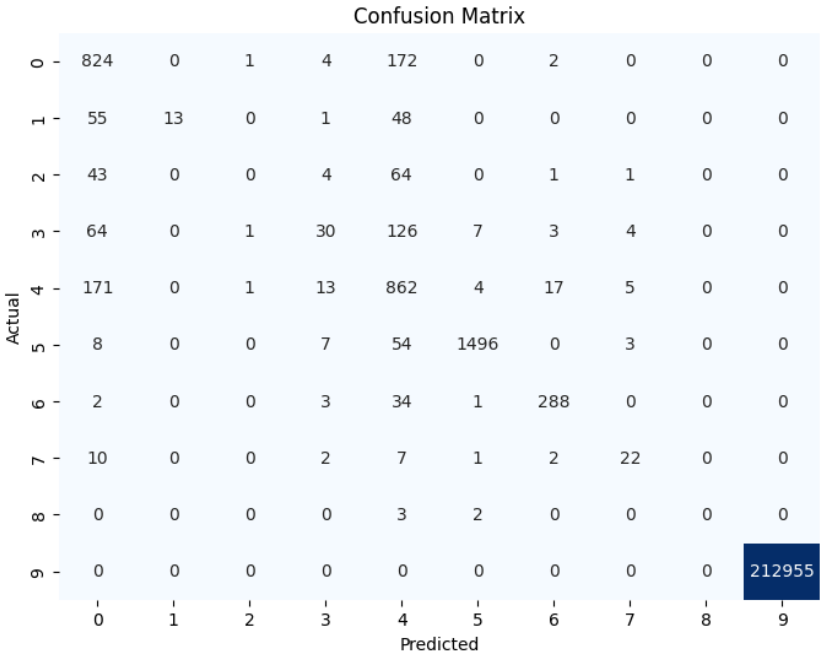


Figure 22. Final results on entirety of dataset.

Our sponsor suggested that we use our random forest model to perform an analysis of the feature importance for each feature, in order to discern which features to remove. This became especially important when we began to train our neural network model to ensure that we achieve the most accurate and unbiased results.

To be able to understand what we are doing and learn while trying to complete this project, our sponsor told us to create a random forest, then a convolutional neural network, then go into graph neural networks. We created a random forest model using sklean with the UNSW dataset. This yielded a result of 90% accuracy. This then led to me trying to understand what is actually resulting in the checks in the random forest.

Talking to our sponsor, he wanted me to remove the IPs from the source and destination, so that the random forest will give results on looking at the data. It prompted me to explore the underlying features and relationships beyond the obvious markers. We found that we had made some errors sorting the data as we included the ids from the CSV.

The id just counts by one in each line. This and removing a few other useless lines led to accuracy of 86% after also removing the IP addresses. One good thing was that we only got false positives, all of the malicious packets were found. Decreasing the false positive rate is the next part of this. We am planning to write a script to remove rows to see how impactful it will be to the accuracy of the random forest model.

We imported a training and test csv given by the sponsor to test our neural network. We separated the label and attack cat for both datasets into a y variable and we trained a random forest neural network using the training csv set. We then used that model to predict the test set. By inspecting the model we were able to look at what features were useful in predicting values. This is important because we can remove the unimportant features for later models to put more emphasis on the more important features.

We also tried using the decision tree to test if it would do better and get different feature importance. It got a slightly lower accuracy and it got different feature importances.

```
def dataPrep(file):
    df = pd.read_csv(file)
    #drops features from the file that the model isn't supposed to use
    X = df.drop(['Label', 'srcip', 'sport', 'dstip', 'dsport', 'attack_cat','ct_state_ttl', 'proto', 'state'], axis=1)
    #compresses 2 features into 1 feature
    X['time_dif'] = X['Stime'] - X['Ltime']
    X = X.drop(['Stime', 'Ltime'], axis=1)
    #sets the target feature
    y = df['attack_cat']

    columnTypes = X.dtypes
    #handles non-integer/string columns
    for key, value in columnTypes.items(): #use one hot encoding
            if value=="object":
                    X[key] = X[key].astype('category')
                    X['new'+key] = X[key].cat.codes
                    enc =OneHotEncoder()
                    enc_data = pd.DataFrame(enc.fit_transform(X[['new'+key]]).toarray())
                    X = X.join(enc_data)
from sklearn.tree import DecisionTreeClassifier
#gets training data from training file
X_train, y_train = dataPrep('ProcessedUNSW-NB15_1.csv')
#declares model object
tree_classifier = DecisionTreeClassifier()
#trains model object on training data
clf.fit(X_train, y_train)
#gets testing data from testing file
X_test, y_test = dataPrep('ProcessedUNSW-NB15_2.csv'))
#inputs test X into the model
y_pred = clf.predict(X_test)
#compares the predicted result from the actual result
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

We removed most of the features because of their low scores. It got a .9 accuracy with most of the error being false negatives, showing that it under predicted anomalies. The features that it prioritizes are not always the most ideal in taking account, but are what the model thinks are the most important. This is demonstrated in other models we ran where we kept useless data in the dataset.

The models would make this useless data a priority, making the model off by a significant amount. This means whether or not each of these features should be used in later models should be tested to see if their absence decreases performance in smaller easier to run models before using them and finding out later.

```
pd.set_option('display.max_columns', None)
def replace_strings(value):
    if isinstance(value, str):
        return 0
    return value
def strip_whitespace(x):
    if isinstance(x, str):
        return x.strip()
    return x
def createCSV(file):
    #uses features file to get the column names
    features = pd.read_csv("NUSW-NB15_features.csv")
    df = pd.read_csv(file)
    df = df.values
    df = pd.DataFrame(df, columns=features["Name"])
    df = df.applymap(strip_whitespace)
    #replaces NAN with a string value
    df['attack_cat'] = df['attack_cat'].fillna("None")
    #keeps backdoor value the same across all values
    df['attack_cat'] = df['attack_cat'].replace("Backdoor", "Backdoors")
    #replaces NAN with integer value
    df['ct_flw_http_mthd'] = df['ct_flw_http_mthd'].fillna(0)
    #keeps column consistantly an integer
    df['ct_flw_http_mthd'] = df['ct_flw_http_mthd'].astype(int)
    #replaces NAN with integer value
    df['is_ftp_login'] = df['is_ftp_login'].fillna(0)
    df['is_ftp_login'] = df['is_ftp_login'].astype(int)
    #gets rid of random spaces
    df['ct_ftp_cmd'] = df['ct_ftp_cmd'].apply(replace_strings)
    nan_count = df
    # Print the result
    df.dropna(inplace=True)
    df.to_csv("Processed"+file, index=False)
def main(argv):
    for file in argv:
        #passes argument processing function
        createCSV(file)
if __name__ == "__main__":
    #takes in file names as arguments
    main(sys.argv[1:])
```

We used the previous lines of code to process data from different files so it could all be uniform. It was given in a format unfit to be turned into a

pandas dataframe because it didn't have the names of the columns in the top row. So, we had to read from a file that had all the column names and assign each of the columns to the corresponding feature name. We also had to fix the data, because different files had different types for the columns, or they had spaces in some of the values but didn't have spaces in other corresponding values in other files.

```
def dataPrep(file):
    df = pd.read_csv(file)
    #drops features from the file that the model isn't supposed to use
    X = df.drop(['Label', 'srcip', 'sport', 'dstip', 'dsport', 'attack_cat','ct_state_ttl', 'proto', 'state'], axis=1)
    #compresses 2 features into 1 feature
    X['time_dif'] = X['Stime'] - X['Ltime']
    X = X.drop(['Stime', 'Ltime'], axis=1)
    #sets the target feature
    y = df['attack_cat']

    columnTypes = X.dtypes
    #handles non-integer/string columns
    for key, value in columnTypes.items(): #use one hot encoding
            if value=="object":
                    X[key] = X[key].astype('category')
                    X['new'+key] = X[key].cat.codes
                    enc =OneHotEncoder()
                    enc_data = pd.DataFrame(enc.fit_transform(X[['new'+key]]).toarray())
                    X = X.join(enc_data)
from sklearn.ensemble import RandomForestClassifier
#gets training data from training file
X_train, y_train = dataPrep('ProcessedUNSW-NB15_1.csv')
#declares model object
clf = RandomForestClassifier(n_estimators=100, random_state=42)
#trains model object on training data
clf.fit(X_train, y_train)
#gets testing data from testing file
X_test, y_test = dataPrep('ProcessedUNSW-NB15_2.csv'))
#inputs test X into the model
y_pred = clf.predict(X_test)
#compares the predicted result from the actual result
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

This code runs random forest and tests it on multiple csv files that were created with the last lines of code. It takes in csv files and puts them through a data prep function which returns an X variable and a y variable, the X variable is used by the model to test against the y variable through

backpropagation. It also drops a lot of the X values that are not supposed to be there because it causes the model to cheat or underperform.

We train the model using one CSV file then we use it to test on three other CSV files. We then feed the results into a sklearn to get a confusion matrix to see where it is failing. We also print out the features it is using to see what the model considers is important, which affects what we focused on for later models.

4.1.15 One Hot Encoding

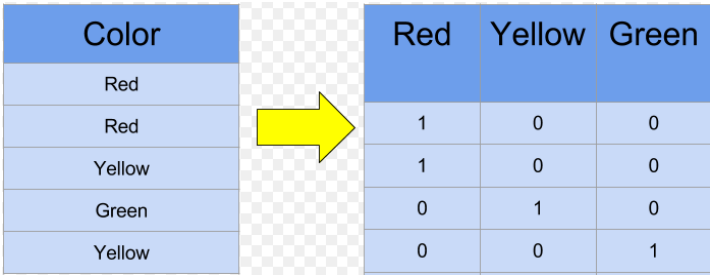
Our sponsor recommended that we use one hot encoder for one of the tasks in our project and since we are unfamiliar with the topic we researched in order to get a foundation and how we could apply it.

Feature encoding is essential in transforming raw data into a format that machine learning models can process. The one-hot encoder is particularly significant in this transformation. It converts categorical variables into a binary matrix representation, enabling algorithms to better understand and predict based on these features.

One-hot encoding involves creating a binary column for each category of a variable. The presence of a category in an observation is marked by '1', while '0' denotes its absence. This method effectively removes the ordinal relationship inherent in numerical encoding, which might not be relevant for all categorical variables.

For instance, consider a dataset with a categorical feature 'Color' having values 'Red', 'Blue', and 'Green'. One-hot encoding will create three

new columns, 'Color_Red', 'Color_Blue', and 'Color_Green', each representing one of these possibilities. [41]



Color
Red
Red
Yellow
Green
Yellow

Red	Yellow	Green
1	0	0
1	0	0
0	1	0
0	0	1

Figure 23. An example of features before and after one-hot encoding. [41]

In our GNN-based anomaly detection project, one-hot encoding plays a pivotal role. The network dataset comprises nodes and edges with categorical attributes, such as types of nodes and relationships. One-hot encoding these attributes allows the GNN to process and learn from these features more effectively.

One challenge encountered was the computational cost associated with encoding a large number of categories, which was mitigated through dimensionality reduction techniques without significantly losing information.

The incorporation of one-hot encoding significantly enhances the GNN's ability to detect anomalies. By accurately representing categorical features, the GNN can discern patterns and deviations more effectively. Preliminary results have shown a marked improvement in the model's ability to identify anomalies in network data.

One-hot encoding is a simple yet powerful technique that plays a crucial role in the preprocessing of data for anomaly detection using GNNs. Through effective encoding of categorical variables, GNNs can better understand and learn from the dataset, leading to more accurate anomaly detection.

4.2 Design Summary

In summary, Graph Neural Networks provide value in the sense that knowledge can be gathered about a specific node's neighbors. Through multiple layers, this data is accumulated so that a node contains not only information about itself but many of its connections. With GNNs for network anomaly detection being a relatively new field, there is limited research in the node and edge anomaly detection and our group looks to provide new insights for dynamic or static graphs.

4.3 Design Description

4.3.1 Tools

4.3.1.1 VS Code

Our preferred Integrated Development Environment (IDE) for this project is Visual Studio Code. VSCode has a large array of plugins that can significantly improve our development workflow. With the use of VSCode and Github, it allows us to enhance productivity and collaborate seamlessly in this project.

4.3.1.3 Github

Our code is hosted using GitHub, an open-source platform for version control. GitHub allows for collaborative coding by hosting a git repository that can be cloned by all team members to develop on their local environment.

One of the main features of GitHub that is new to all team members is GitHub Enterprise. GitHub Enterprise allows you to be a part of an organization and collaborate with other organization members. This means that instead of working strictly on one project, organization members can collaborate on multiple projects at a time. Due to our sponsor's affiliation with the Georgia Tech Research Institute (GTRI), all team members were granted an enterprise account to connect with other GTRI members. Two of the repositories set up for the project are our Resources repository and our Random Forest Pipeline repository. Having both of these under the same scope with GitHub enterprise makes collaborative work much easier.

In order to reach our goal product, we need to heavily test our code and can leverage GitHub Actions to do so. GitHub Actions allows users to define some workflow and then perform it based on some trigger. Although most group members have experience with Continuous Integration and Continuous Deployment, it was with other platforms such as Buildkite or Jenkins. We look to explore GitHub actions when pushing code that is going to be heavily used in our network simulation to GNN pipeline. Aside from testing, we can ensure that we are always using the latest simulation pipeline code for Pacific Northwest National Laboratories by cloning the latest version before running a simulation.

A feature of GitHub that was initially requested by our sponsor was GitHub issues. GitHub issues allow developers to track their progress and connect their tasks to pull requests. We decided to use Jira instead of GitHub issues for this project.

4.4 Production Plan

Once we completed our literature review process, we needed to brainstorm any extensions to this paper that can improve the neural network. During this process, we needed to plan our own Graph Neural Network in detail and come up with an outline of what we will expect to get accomplished for the next semester. This includes code to be written and what experiments we plan to perform and run. The following production plan outlines the goals our team planned for Senior Design 2 during our Senior Design 1 semester. Some of these goals needed to be adjusted as we ran into different challenges with implementation and experimentation.

It is one of my team's goals to have this simulation running within the next month. This will give us the opportunity to inspect our data and make important design decisions based on the problem scope. Our planning should be done by the end of the semester.

Since we do not have enough background knowledge yet to begin programming our Graph Neural Network, we planned with our sponsor to use some basic models in order to get an overview of the problem in the meantime. As previously mentioned, each group member has individually coded up and trained a random forest model. We are beginning to compare

our results and planning to use the random forest to perform a preliminary feature importance analysis.

This will help us to gain more accurate results on the random forest model, as well as the traditional neural network model. Our neural network model will most likely consist of a traditional multilayer perceptron network, and our sponsor has requested that this (along with our GNN) be coded using the Pytorch library [20][42]. The group will begin working together on the neural network model and we plan to present our results to our sponsor at the end of the semester.

Following the completion of these initial steps, the subsequent phase involves delving into the practical implementation of the Graph Neural Network (GNN) and rigorously testing our model. This critical stage not only allows us to validate the efficacy of our approach but also provides valuable insights into the performance of the GNN in the context of our senior design project. At the end of the senior design project, we should have a report or paper written about our final results. As a stretch goal, it would be a fantastic opportunity for our entire team to turn our project into a published paper or part of a paper with the help of our sponsor.

4.5 Setup and Running Instructions

The project source code is stored in the GitHub repository https://github.gatech.edu/GTRI-UCF-Senior-Project-2023-2024/UNSW-experiments along with detailed documentation, setup, and running instructions. This repository is currently private within the Georgia Tech organization, but we are working with our sponsor to make this information public.

5. Conclusions

5.1 Characterization of Results

In our initial phase we took a deep dive into the dataset, and gained a strong understanding of the nuances and potential implications for anomaly detection. We identified important features and patterns that would classify specific anomalies. The process helped determine preprocessing steps needed to make the data be able to be used in our code/scripts. It was also helpful in becoming more familiar with the data overall.

One task provided by our sponsor that we have completed is creating a PyShark script that transforms Packet Capture files into a structured comma separated-value format. This script is an important component of the data pipeline we are using. It allows us to convert raw network traffic data into a format that we can read as well as be used by machine learning models that we are also using for our network. The script extracts the key features we need efficiently which makes sure that our models have relevant and clean inputs.

After understanding our data and its features we started to develop models that will help build a foundation, starting with the creation of random forest models. These models serve as an initial testing environment by providing us with a baseline understanding of the predictive potential of the data we are using. We trained the random forest models on our preprocessed dataset, and we were provided with initial insights in the behavior of the data and highlighted potential areas of focus for the graph neural network. Although the results from these models were preliminary

they still provided value by demonstrating the usefulness of machine learning for anomaly detection in network data.

Another important component in our project that we were able to handle was establishing computational infrastructure. We have successfully set up a Docker environment, which allows us to run simulations in a consistent and isolated setting, as our project grows we were able to scale our project however we needed. This environment has many other advantages, like streamlining the process of managing different dependencies and system configurations, which will get rid of the need to set up multiple development environments. Most importantly setting up Docker helps ensure control and consistency by allowing our group members to collaborate seamlessly.

Along with the extensive amounts of research put into understanding topics relating to our project during the first semester of senior design, we have successfully obtained results on the UNSW (CITE) dataset using a Graph Neural Network with the GraphSAGE implementation to predict attack presence and category with an accuracy of approximately 98% on average over multiple training sessions. We have obtained results for not only a model that categorizes between attacks and non-attacks, but also a model that will categorize attack type. Our results from this project have served as a proof of concept that Graph Neural Networks are a suitable option to use for anomaly detection on the cybersecurity problem domain.

Over the course of this project, our team learned a lot about not only GNNs, but also about project management, research collaboration, and the AI development process. It is our hope, along with our sponsor, that future senior design groups will continue this work and add to our work. Some

options for future development include creation of a novel GNN, improving our GNN performance by tweaking hyperparameters, testing on the simulation data, integrating the data pipeline into one application, and providing an informative front-end GUI.

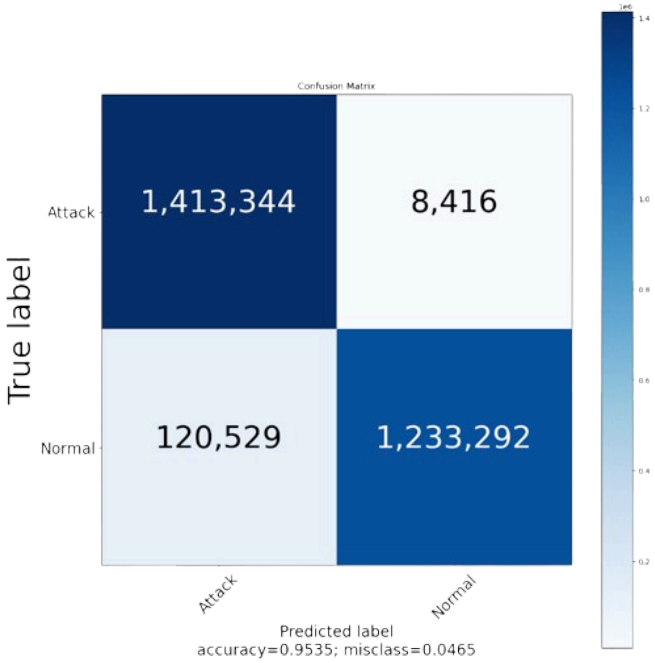


Figure 24. An illustration of accuracy results for our initial GNN model.

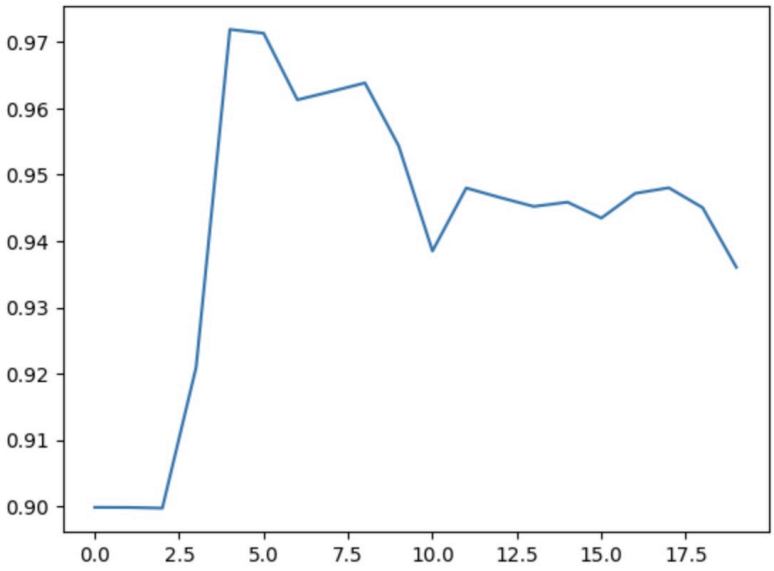


Figure 25. An illustration of accuracy during training for our final GNN model.

Note: We saved and tested the model with best accuracy over all epochs of training.

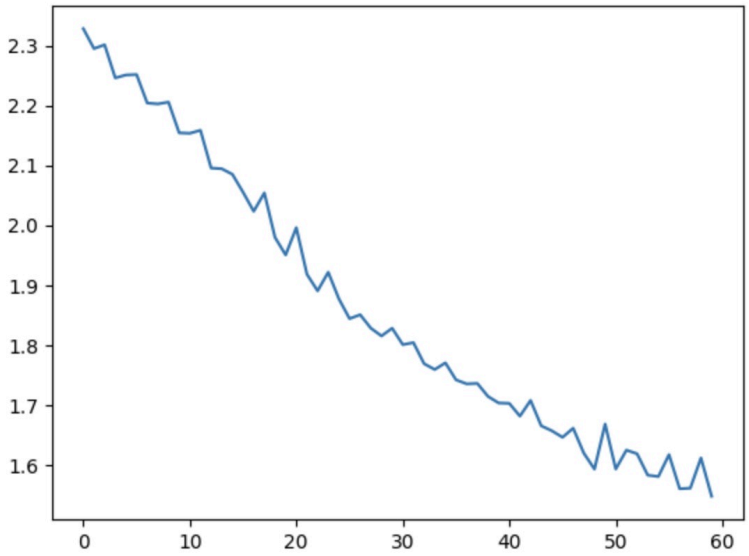


Figure 26. An illustration of loss during training for our final GNN model.

5.2 Summary of Project

In order to implement the Graph Neural Network we used the pytorch-geometric library. This library uses tensors to represent the node and edges of the graph. Tensors are multi-dimensional arrays that have easier numerical computation in the GPU when compared to normal multi-dimensional arrays, making them ideal for neural networks due to the large amount of computations needed to train and run the model.

Edges are represented by two tensor arrays, the first array representing from what node and the second array representing to what node the edge is going. Nodes are represented as a two dimensional array, the first dimension representing the nodes and the second dimension representing features. The library can also represent multiple types of nodes through a dictionary of node tensors, the types of nodes differing in the amount of features that they each respectively have

The first step in implementing a Graph Neural Network using the pytorch-geometric library is formatting the packet capture (pcap) data in such a way that it can fit in a graph. The main issue with this is that there are many ways in which this can be done, and this greatly affects the effectiveness of the Graph Neural Network. The most intuitive solution to this could be to have the nodes represent IP addresses and the edges represent data being shared between the IP addresses.

The properties of the edges between nodes cannot be the same as the packet capture data because there can be many requests made between two

nodes, so the properties of the edges between two nodes must be able to represent many requests.

The Graph Neural Network would predict what IP addresses are producing anomalies using node classification. The program would then mark any of the packet captures as anomalies that share an IP address with the nodes that the Graph Neural Network determined to be anomalies.

Overall, our minimum viable product for this project is to apply a graph neural network to the problem of cybersecurity attack in the United States Smart Power Grid using simulated data. As previously mentioned, our group has been working up to this goal by taking a look at the existing research in this area. We have also taken steps towards this goal by analyzing a preexisting, simpler dataset which does not require us to simulate it. This preliminary analysis has been done using some simple models in various python libraries. By becoming familiar with this type of data and the data formats, we will be able to make more informed decisions about the structure of our final graph neural network product.

As our final goal, our group would like to create an extension on a preexisting graph neural network structure, improving on the state of the art and adding to existing literature. Our coding work would most likely involve a reproduction of some previous model and a modification of this program in order to create a new and improved type of graph neural network model. This will be tested on simulated data, using a state of the art simulation provided by our sponsor.

Our stretch goal is to produce an academic paper on the topic of graph neural networks from the perspective of a smart power grid cybersecurity attack. This would involve testing our code, comparing it to the state of the

art, and composing a solid analysis of our findings, adding to the current academic literature on the topic of graph neural networks.

6. Administration

6.1 Expectation Outline

Each team member is expected to attend every meeting with exceptions to erroneous circumstances. All the members should be expected to do similar amounts of work and be accountable for their work. All other members should check other members' work. Meetings will start by taking attendance, and a member will write the meeting minutes. The chosen member will be in a rotation per meeting. The members of the group should motivate each other to get through the project.

For communication, we will be using Discord primarily. On the discord, we have created multiple channels to talk about specific topics. This makes it organized so that we go back and look at our research.

During the Literature Review, each team member is expected to read all of the papers included in our Literature Review. Each team member is expected to be able to use GitHub proficiently. They are also expected to make push requests at regular intervals.

Our sponsor has requested that we name a team captain and that our team not use Agile Methodology. We have decided together that Emily Hannon will be the team captain. This role will be a responsibility in addition to the rest of the project. The team captain will be responsible for ensuring that the team agrees on meeting times, deadlines, and assigned work and for communicating effectively with the sponsor.

Instead of using Agile Methodology, our team structure will resemble a more traditional structure of a research team. Our sponsor will preside over our group meetings as our research mentor and the rest of the team will be

expected to contribute to the project as equals. Each team member will be expected to contribute the same amount of work (writing, code, background reading, etc.) during the entirety of our project.

In terms of accountability, we will show leadership qualities to our other team members in order to set a good example. We will understand each other's capabilities and knowledge to set clear expectations.

As our project mainly focuses on research, most of the first semester will be spent learning and understanding our project as well as where other projects have fallen short. Communication will happen over our team Discord channel. When it comes to implementation and coding, we plan on using a Discord bot to automate the process of our daily standup. This will allow us to have transparency and effective communication on a frequent basis.

There will always be a due date/time for tasks so that all team members will know when something is going to be due. All coding deliverables must be pushed to our GitHub by the due date/time in order to be considered complete.

It is important to outline the difference between a lack of understanding and a lack of effort. In the case that someone is lacking understanding, we will make an effort as a team to communicate with them and provide them resources to learn. This will not require any further action because we are all here to learn. If someone is constantly putting in significantly less effort than expected, everyone else on the team will have a talk with them and outline specific expectations moving forward. We plan on talking to Leinecker and/or Gerber and then move to our sponsor if the behavior does not change.

To prepare for unexpected circumstances that impact a member's ability to complete their task, communication will be a requirement. When it comes to progress in a deliverable, code, etc. each member will know where everyone is at in their respected task. This will allow the team to decide how they will adjust to remedy the situation in the event of a member having an unavoidable issue that places a halt in their ability to contribute to their goal.

If this does happen, the team will create a plan to work around the problem and decide what needs to be done. Whether it comes to a different member filling in to complete a necessary deliverable, or multiple members working together, the team will come together to discuss a plan on how to proceed. Relative to the emergency, the team will decide if we need to contact our TA’s, Professors, and Sponsor in order to communicate the issue.

In the case of team conflicts, each member will be expected to communicate with the team if it cannot be resolved and come together as one. All voices will be considered when an issue occurs, and the main goal will be to handle the situation as quickly and respectfully as possible. Ensuring that no bad blood lingers will be paramount in continuing to have strong team dynamics throughout the project. As before, if the conflict cannot be resolved by the team, the team will decide if we need to contact our TA’s, Professors, and Sponsor in order to resolve the issue.

Finally, if a team member is consistently unable to deliver the agreed-upon set of deliverables, the team will meet to reassess responsibilities carried by individual team members and the team as a whole following the interest of the team and project.

6.2 Product Backlog

Item ID	Task	Priority	Status
GNN-2	PCAP Parsing	Medium	Done
GNN-3	UNSW CSV Files Analysis	High	Done
GNN-4	Random Forest on UNSW	High	Done
GNN-5	Supervised Neural Network on UNSW	High	In Progress
GNN-7	Run data simulation for NATIG dataset	Medium	To-Do
GNN-8	Setup Docker Sim Environment	Medium	Done
GNN-9	Integrate Docker with PCAP	Low	To-Do
GNN-13	Sim PCAP Random Forest	Low	To-Do

6.3 Milestones

Milestone	Description	Responsibilities	Date
First Group Meeting	Introductions and discussed plans for the project.	All team members	9/29/2023
First Sponsor Meeting	Went over hosting and starting resources.	All team members	9/29/2023
Design Proposal and Team Contract	Wrote our individual thoughts regarding the project and reviewed team expectations.	All team members	10/6/2023
Graph Neural Networks	First lesson on GNNs. Drew diagrams to explain the concept of message parsing.	Team discussion led by Santiago Rodriguez	10/6/2023
TA Check-In #1	Went over how to work on a project	All team	10/13/2023

	that is more research than development. Discussed how to designate team roles.	members	
Initial Jira Setup	Added stories to our Jira under Epics. Explained Jira to other unfamiliar members.	Emily Hannon	10/16/2023
UNSW Experiments GitHub Repo Created	Created a repository within Georgia Tech Research Institute (GTRI) GitHub enterprise to host our Random Forest and basic neural network code.	Emily Hannon	10/16/2023
Initial Random Forest	Wrote Python code on Random Forest with 95% accuracy to perform the binary classification of	Nicholas Lannon	10/23/2023

	whether an attack occurred based on packet data.		
Check-In with Leinecker	Went over project updates. Discussed the application of Jira and Agile on a research-heavy project.	All team members	10/23/2023
Docker Simulation Environment Setup	Set up a simulation environment with Docker from NATIG Repo. This environment lets us gather simulated packet data.	Mukundh Vasudevan	11/8/2023
Autoencoder	Wrote an autoencoder in Python to perform the same task as the Random Forest. If an attack is predicted,	Gustavo Nazario Perez	11/12/2023

	this will trigger a Random Forest model to predict the attack type.		
PCAP Parser	Wrote a script using pyshark to parse packet data and convert it to a CSV file. This will be used in our final Graph Neural Network pipeline.	Landon Russell	11/15/2023
TA Check-In #2	Discussed status of final design document and the short and long-term goals of the project heading.	All team members	11/17/2023

6.4 Finances

There is no budget allocated to this project as there are no necessary expenses. Our group plans to utilize the Newton cluster at UCF [43] in the

case where any additional computational resources are required. This may be necessary for running the simulated data from the Network Attack Testbed in [Power] Grid simulation. [3]

7. Acknowledgements

7.1 Sponsor Assistance

The sponsor for this project is Branden Stone, a Research Scientist at Georgia Tech Research Institute. His research revolves around developing original artificial intelligence and machine learning technologies. The team and sponsor met on a weekly basis. Mr. Stone constantly provided the team with the necessary resources to succeed. Some of these resources include Papers With Code and a Graph Representation Learning Book [10]. He allowed the team to ask questions whenever they needed to about anything related to the project.

7.3 Bibliography

[1] H. Kim, B. S. Lee, W. -Y. Shin and S. Lim, "Graph Anomaly Detection With Graph Neural Networks: Current Status and Challenges," in IEEE Access, vol. 10, pp. 111820-111829, 2022, doi: 10.1109/ACCESS.2022.3211306.

[2] Oceane Bel, Joonseok Kim, William J Hofer, Manisha Maharjan, Sumit Purohit, & Shwetha Niddodi. (2023). Co-Simulation Framework For Network Attack Generation and Monitoring.

[3] Moustafa, Nour. Designing an online and reliable statistical anomaly detection framework for dealing with large high-speed network traffic. Diss. University of New South Wales, Canberra, Australia, 2017.

[4] U.S. Department of Energy, Office of Electricity. (2019, December 16). Smart Grid: The smart grid. Smart Grid: The Smart Grid | SmartGrid.gov. https://www.smartgrid.gov/the_smart_grid/smart_grid.html

[5] United States Government. (2016). (rep.). Cyber Threat and Vulnerability Analysis of the U.S. Electric Sector. Retrieved November 2023, from https://www.energy.gov/policy/articles/cyber-threat-and-vulnerability-analysis-us-electric-sector.

[6] United States Government. (2023, October 20). Frequently asked questions (faqs) - U.S. energy information administration (EIA). Frequently Asked Questions (FAQs) - U.S. Energy Information

Administration (EIA). https://www.eia.gov/tools/faqs/faq.php?id=108&t=3#:~:text=In%202022%2C%20U.S.%20electric%20utilities,electric%20meters%20were%20AMI%20meters

[7] U.S. Energy Information Administration. (2022). Advanced Metering Count by Technology Type. SAS output. https://www.eia.gov/electricity/annual/html/epa_10_05.html

[8] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, & Koray Kavukcuoglu. (2017). Population Based Training of Neural Networks.

[9] A. Daigavane, B. Ravindran, and G. Aggarwal, "Understanding Convolutions on Graphs," Distill, vol. 6, no. 9, p. e32, Sep. 2021, doi: 10.23915/distill.00032.

[10] Faroz, S. (2022, September 14). Geometric deep learning with graph neural network. Medium. https://salmanfaroz.medium.com/geometric-deep-learning-with-graph-neural-network-ace43692622f

[11] Hamilton, W. L. (2020). *Graph Representation Learning*. Springer International Publishing. https://doi.org/10.1007/978-3-031-01588-5

[12] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko, "A Gentle Introduction to Graph Neural Networks," Distill, vol. 6, no. 9, p. e33, Sep. 2021, doi: 10.23915/distill.00033.

[13] William L. Hamilton. 2020. Graph Representation Learning Book. Mcgill University.

[14] Yulia Kosarenko. (2021). How To Create Decision Trees for Business Rules Analysis. Why-change. https://why-change.com/2021/11/13/how-to-create-decision-trees-for-business-rules-analysis/

[15] S. Rabanser, O. Shchur, and S. Günnemann, "Introduction to Tensor Decompositions and their Applications in Machine Learning." arXiv, Nov. 29, 2017. doi: 10.48550/arXiv.1711.10781.

[16] X. Ouvrard, J.-M. L. Goff, and S. Marchand-Maillet, "Adjacency and Tensor Representation in General Hypergraphs Part 1: e-adjacency Tensor Uniformisation Using Homogeneous Polynomials." arXiv, May 30, 2018. Accessed: Nov. 28, 2023. [Online]. Available: http://arxiv.org/abs/1712.08189

[17] S. Zhou, Q. Tan, Z. Xu, X. Huang, and F. Chung, "Subtractive Aggregation for Attributed Network Anomaly Detection," in Proceedings of the 30th ACM International Conference on Information & Knowledge Management, in CIKM '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 3672–3676. doi: 10.1145/3459637.3482195.

[18] L. Huang et al., "Hybrid-Order Anomaly Detection on Attributed Networks," IEEE Trans. Knowl. Data Eng., pp. 1–1, 2021, doi: 10.1109/TKDE.2021.3117842.

[19] Singh, P., P, J. J., Pankaj, A., & Mitra, R. (2021). Edge-detect: Edge-centric network intrusion detection using Deep Neural Network.

2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC). https://doi.org/10.1109/ccnc49032.2021.9369469

[20] Pyg Documentation. PyG Documentation - pytorch_geometric documentation. (n.d.). https://pytorch-geometric.readthedocs.io/en/latest/

[21] Grubbs, F. E. (1974). Procedures for detecting outlying observations in samples. Defense Technical Information Center.

[22] Ma, X., Wu, J., Xue, S., Yang, J., Zhou, C., Sheng, Q. Z., Xiong, H., & Akoglu, L. (2023). A comprehensive survey on graph anomaly detection with deep learning. IEEE Transactions on Knowledge and Data Engineering, 35(12), 12012–12038. https://doi.org/10.1109/tkde.2021.3118815

[23] Z. Zhang, Y. Li, W. Wang, H. Song, and H. Dong, “Malware detection with dynamic evolving graph convolutional networks,” Int. J. Intell. Syst., vol. 37, Mar. 2022, doi: 10.1002/int.22880.

[24] Sanderson, “But what is a Neural Network?” Accessed: Oct. 27, 2023. [Online]. Available: https://www.3blue1brown.com/lessons/3blue1brown.com

[25] Representations for Social Recommendation School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China

[26] Maxim Liu. (2021). What is the Autograd? Pytorch Design Patterns Explained(1) - Autograd. Medium.

https://maximliu-85602.medium.com/what-is-the-autograd-pytorch-design-patterns-explained-1-autograd-5320cbed8cb3

[27] Guatam Ethiraj. (2022). What is nn.Embedding Really?. Medium. https://medium.com/@gautam.e/what-is-nn-embedding-really-de038baadd2

[28] Yulia Kosarenko. (2021). How To Create Decision Trees for Business Rules Analysis. Why-change. https://why-change.com/2021/11/13/how-to-create-decision-trees-for-business-rules-analysis/

[30] Bhuvaneswari Gopalan. (2020). What is Gini Impurity? How is it used to construct decision trees? Numpyninja.

[31] Viswateja. (2019). Measure of Impurity. Medium. https://medium.com/@viswatejaster/measure-of-impurity-62bda86d8760

[32] Carbonati. (2016). Random Forest From Scratch. GitHub. https://carbonati.github.io/posts/random-forests-from-scratch/

[33] S. Zhang, H. Tong, J. Xu, and R. Maciejewski, "Graph Convolutional Networks: A Comprehensive Review," Computational Social Networks, vol. 6, no. 1, p. 11, Nov. 2019, doi: 10.1186/s40649-019-0069-y.

[34] How powerful are graph convolutional networks?. Thomas Kipf. (n.d.). https://tkipf.github.io/graph-convolutional-networks/

[35] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs." arXiv, Sep. 10, 2018. doi: 10.48550/arXiv.1706.02216.

[36] Ahmed, F., Cui, Y., Fu, Y., & Chen, W. (2021). A Graph Neural Network Approach for Product Relationship Prediction. ArXiv, abs/2105.05881.

[37] Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio', P., & Bengio, Y. (2017). Graph Attention Networks. ArXiv, abs/1710.10903.

[38] Michaël Defferrard, Xavier Bresson, & Pierre Vandergheynst (2016). Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. CoRR, abs/1606.09375.

[39] Keyulu Xu, Weihua Hu, Jure Leskovec, & Stefanie Jegelka (2018). How Powerful are Graph Neural Networks?. CoRR, abs/1810.00826.

[40] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, & George E. Dahl (2017). Neural Message Passing for Quantum Chemistry. CoRR, abs/1704.01212.

[41] Dansbecker. (2018, January 22). Using categorical data with one hot encoding. Kaggle. https://www.kaggle.com/code/dansbecker/using-categorical-data-with-one-hot-encoding

[42] PyTorch. (n.d.). https://pytorch.org/

[43] About Newton. Home. (2023, September 24). https://arcc.ist.ucf.edu/index.php/resources/newton/about-newton