Training an unsupervised GNN model for load balanced clustering of IoT networks

Dănilă, Andrei

Award date:
2022

Department of Mathematics and Computer Science
Interconnected Resource-aware Intelligent Systems Research Group

Training an unsupervised GNN model for load balanced clustering of IoT networks

Master Thesis

Andrei Danila

Supervisors:
Dr. T. Ozcelebi
Dr. Q. Liu
Dr. C. Güven

Committee:
Dr. T. Ozcelebi
Dr. Q. Liu
Dr N. Zannone

Eindhoven, July 2022

Abstract

Internet of Things (IoT) networks have become common place in the modern world. They are present everywhere, from houses, to industry, to fields and greenhouses. All of these interconnected devices suffer though from several issues. The most discussed of which is, possibly, battery consumption. Due to the fact that most IoT devices are small, cheap and deployed in a multitude of places, they are battery powered. However, batteries need to be replaced eventually. The problem then becomes that the process of replacing batteries is expensive, time-consuming or even unfeasible in some cases. As a consequence of this, research is constantly being done to attempt to prolong the lifetime of devices. One technique to achieve longer battery life is to balance the load suffered by devices because of network traffic.

In the IoT infrastructure, a middle layer between the end devices and the cloud is represented by edge servers. These nodes benefit from higher computational power than normal devices and, possibly, even a wired power connection. This aspect makes them suitable for organizing the network in clusters around them. There is a need to balance the load of these clusters to prolong the lifetime of the network. The clustering would organize the traffic of the network, and by also balancing the load of these clusters, it ensures a more even consumption of energy across the network.

In this thesis, an unsupervised Graph Neural Network (GNN) model is proposed and trained to tackle the issue of load balancing the clusters formed around the edge servers. Our model tries to deal with the problem by deciding the cluster membership of individual nodes. The training is done such that the cluster unity is maintained in the pursuit of better load balance between clusters. The training and testing are done without any labelled data, removing the normal need for the extensive task of gathering and manually labelling the data. Results prove that our model clusters IoT networks of varying sizes and topologies the best overall.

Acknowledgements

I would like to thank my supervisors for helping and guiding me through this master project. They have provided valuable insight and feedback.

I would like to thank the members of my committee for agreeing to review and grade my work.

I would like to thank my parents that have always supported me both emotionally and monetarily regardless of the situation.

Lastly I would like to thank my girlfriend, without whose support and encouragement it would not have been possible to finish the thesis successfully.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Internet of Things (IoT) is the general concept that describes the sea of connected devices that have become ubiquitous in the modern day. These devices can range from light bulbs to fridges [1]. According to Gurunath et al. [2], the current number of IoT devices could be around 60 billion devices, making it a huge market of possibility but also concern [3]. All of these "intelligent" devices follow the same main concepts of availability, being constantly connected to the Internet, and having a higher level of interaction than their "dumb" counterpart.

One of the prospective uses of IoT is for monitoring tasks, for example in the agriculture sector, wildlife monitoring, or smart building surveillance. The use of IoT monitoring in the agriculture domain especially has seen a huge rise in popularity, with a lot of attention being focused on this field [4]. However, one common aspect that all IoT applications have is the concern for battery life. While most IoT devices are small, inexpensive, and efficient, they are still battery-powered, which provides both advantages and disadvantages. The biggest advantage is fast deployment without the need for laying down infrastructure. This allows for covering large areas of space, even if the terrain is unfavourable. The main disadvantage is the power concern. Batteries will eventually run out, and the operation of retrieving and replacing them might be time-consuming, for instance, in the case of a large number of devices, or even unfeasible, in the case of dangerous environments. To this end, power consumption of IoT devices is constantly a hot topic for research [5] [6] [7], especially since this problem can be approached from so many angles.

In the world of wireless networks, there can be multiple ways to set up a network, depending on the wireless communication protocol chosen, the use case, the environment, etcetera. A popular type of network in the IoT space is the multi-hop mesh network [8]. Each device has a limited communication range, and to achieve large distance communication, the message is passed from neighbour to neighbour until it reaches its destination. The complexity of such a setup can increase rapidly with the number of devices in the network, creating congestion and the need for retransmission, which can lead to increased energy consumption. In these cases, a more advanced form of organisation, such as clustering, can prove beneficial.

One of the simplest forms of organisation is clustering. Clustering at its core can be described as a way to segregate items and can be accomplished in a multitude of ways depending on the defined criteria. In the case of wireless networks [9] [10], it serves as a way to split the network into subnetworks. This split can be either logical based or performance based. A logical split means that all devices with the same purpose are in the same cluster. A performance split means devices are split into clusters to achieve better performance of some kind. Such a performance metric can be load balancing [11] [12]. Creating load balanced clusters is one of the easier concepts to understand. The network is split into clusters such that each cluster has roughly the same load as every other cluster, thus ensuring some form of balance. The criteria under which this load balancing is achieved can vary wildly depending on the context, number of clusters, how the load

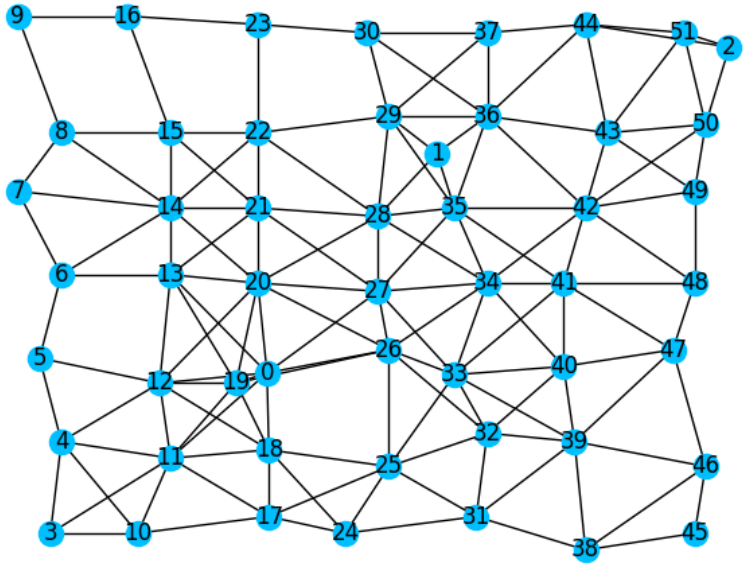

Figure 1.1: Example IoT mesh network

is defined, etcetera.[13] The better distribution of load across the network has a direct beneficial impact on the network lifetime.

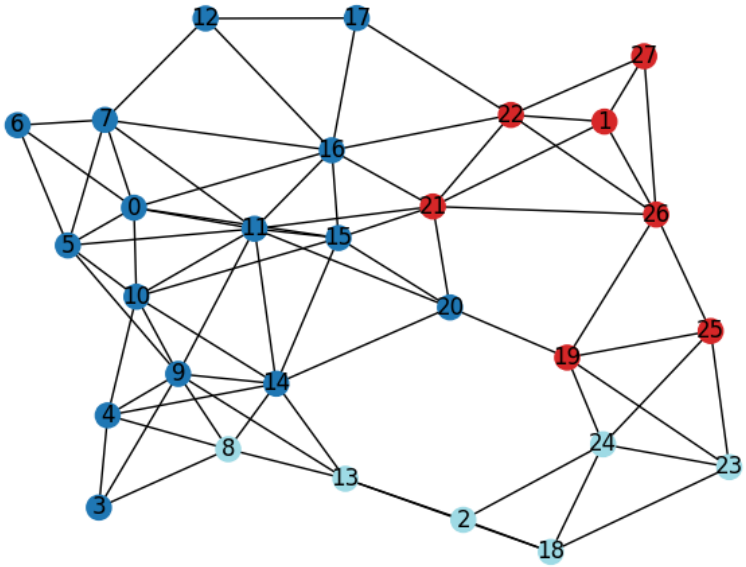

Figure 1.2: Example clustered IoT mesh network

There are multiple ways in which clustering can be achieved [10]. One such method is the use of machine learning. Machine learning, a method to achieve artificial intelligence, has become a highly popular field of research. The reason why is that it promises approachable solutions for a multitude of problems, from image classification to text creation. The possibilities of machine

learning are virtually boundless. A subset of machine learning that has gained recent attention is graph neural networks (GNN). GNN uses graph structures as an input for the machine learning process. GNN has seen use in many fields since graphs are an attractive data structure to represent a multitude of relations, from molecules to social interactions [14]. GNN is particularly appealing for research in the field of wireless networks since networks can easily be interpreted as a graph structure [15] [16]. Applying GNN on the issue of clustering a wireless (IoT) mesh network for the purpose of load balancing is a worthwhile venture.

1.2 Research questions

This thesis sets out to answer the following research questions:

- **What unsupervised GNN method can be used to achieve the best load balance clustering in a wireless mesh network?**
- **What is the best load balancing performance of using GNN for clustering on a wireless mesh network?**
- **How well does the proposed unsupervised GNN approach deal with dynamic events in the network?**

1.3 Goals

This thesis proposes an unsupervised GNN based approach to clustering a wireless network, such that load balance is achieved between the clusters.

Each wireless network differs from one another in multiple ways. However, some of the most important ways are the topology of the network and the traffic of each node. Given the multitude of networks that can exist, obtaining labelled data in this field can be very costly and time intensive. GNN has mainly been used in supervised or semi-supervised ways due to the generally higher achieved prediction accuracy for such methods.[17] However, for the reason mentioned earlier, unsupervised learning might be the better choice for this problem context. This is true twofold for dynamic networks where the size of the dataset increases exponentially with time due to the addition of time variations. To this end, we propose an unsupervised way of leveraging both the structural information the graph structure gives and the network information that is generated through the use of the simulation environment.

GNN provides the backbone of the implementation. Given a certain feature set representation of the network along with the graph representation of the network, the GNN will output the new clustering of the network. The innate advantage of GNN is that the model's structure is dictated by the number of classes and size of the feature set. This allows an approach that can be applied on different topologies as long as the number of server nodes in the network stays the same without additional training. Each cluster contains one and only one server node, which shows that our approach has the possibility to be highly adaptable.

The combination of unsupervised learning and GNN aspects gives our approach a high degree of flexibility. Since the training process is not restricted by labelled data, our approach can be used in multiple scenarios depending on how the GNN is trained. It can become a more general purpose solution if trained on a wide variety of networks or can become more specialised if trained on a dataset of similar networks.

To sum up, the main contributions of this thesis are as follows:

- We propose an unsupervised GNN-based approach to cluster wireless networks while attempting to load balance the clusters.
- Based on the GNN solution, we analyse the viability for use on dynamic networks.
- We perform an empirical study of the approach on synthetic data.

1.4 Thesis layout

The thesis is organised as follows:

- Chapter 2 presents in detail the problem we are trying to solve.
- Chapter 3 describes the technologies used in this project.
- Chapter 4 presents the research papers used in this project
- Chapter 5 presents our GNN solution for the problem.
- Chapter 6 describes the experimental setting and the obtained results of our approach on the synthetic data.
- Chapter 7 concludes the thesis and presents the possible future work on this project.

Chapter 2

Problem statement

2.1 General description

Consider a wireless mesh network of IoT devices comprised of sensor nodes and a few server nodes. A communication link is formed between any two nodes if each one of them is within the transmission range of the other. The IoT nodes acquire data with various sensors from the physical environment and periodically send the sensing data to one of the edge servers in multi-hop communication. The nature of the sensor data is irrelevant for the purposes of this project. Sensor nodes can both receive messages and send messages. They send their own data and forward the data of other sensor nodes toward the destination server node. The number of clusters is equal to the number of server nodes, and a cluster is defined by its single server node and a subset of sensor nodes. The server nodes act as sinks for the rest of the nodes in their respective cluster. The server nodes could be further connected to some form of cloud storage, which is beyond the scope of this project.

A network manager in the network dictates how and when nodes communicate. The network communicates under TDMA, meaning that time is split into timeslots. During a timeslot, each node can either send data or receive data, but not both.

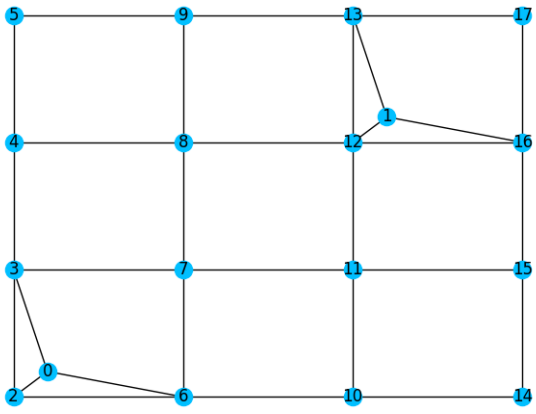

Figure 2.1: Example grid shaped network

The deployment of the nodes can be done either in an organised fashion or in a pseudo-random manner. We assume there is a minimal distance between nodes such that they do not overlap. Additionally, the created network must not become disjoint due to the placement of nodes. Overall, this pseudo-random manner should ensure a realistic representation of real-world

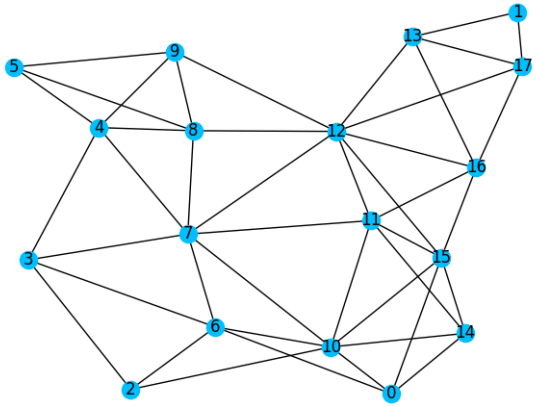

Figure 2.2: Example pseudo-random deployed network

networks. Additionally, all IoT nodes are assumed to be identical to remove any variability. The server nodes are assumed to be connected to a power outlet. All IoT nodes are able to move.

The multi-hop communication is done according to the shortest path to the destination. As a consequence of this, the load of nodes in the network is naturally unbalanced. Nodes that are closer to the server will inevitably have a higher load than nodes further away, which creates a strong relation between the position of a node and its load. Additionally, this means that the load of each server is unbalanced depending on the network topology. This unbalance can lead to congestion of certain nodes, which results in higher latency and more energy consumed due to waiting.

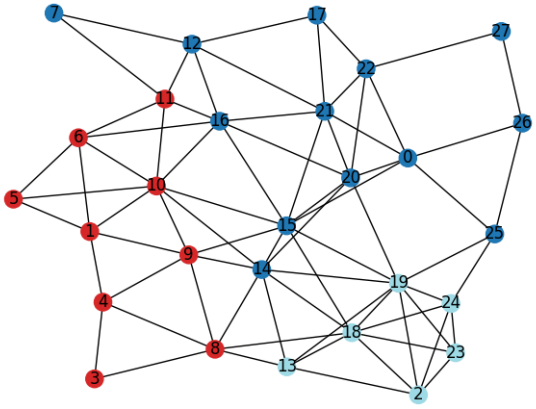

Figure 2.3: Unbalanced clusters

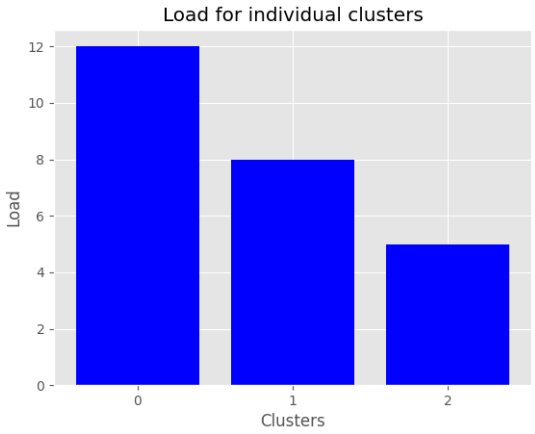

Figure 2.4: Unbalanced loads

2.2 Notation

In this chapter, the formal notation of terms will be presented to help understand the formulas in the rest of the thesis.

2.2.1 Network description

N: set of sensor nodes, $N \subset \mathbb{N}$
$|N|$: number of sensor nodes in the network
C: set of clusters, $C \subset \mathbb{N}$
$|C|$: number of clusters in the network
S: set of server nodes, $S \subset N$
$|S|$: number of server nodes in the network
T: set of timeslots $T \subset \mathbb{N}$
$|T|$: number of timeslots
E: set of links in the network, $E \subseteq N \times (N \cup S)$
n: an sensor node in the network ($n \in N$)
c: a cluster ($c \in C$)
s: a server node ($s \in S$)
t: a timeslot ($t \in T$)
$G(N \cup S, E)$: the graph formed from the node set N, the server set S and the link list E
A: the adjacency matrix of graph G
p: the number of seconds between which a sensor node must report its own data
(n, n'): a link in the network, between nodes n and n', $(n, n') \in E$, $n' \in N \cup S$
$nodes(c)$: the number of nodes in cluster c, $nodes(c) : C \to \mathbb{N}$
$cluster(n, c)$: returns 1 if node n is in cluster c, 0 otherwise, $cluster(n, c) : N \times C \to \mathbb{B}$
$config(n)$: the clustering configuration of node n. $config(n) =$

$$\{cluster(n, c_1), cluster(n, c_2), ...cluster(n, c_{|C|})\}$$

$hop(n, s)$: the minimum number of hops needed for node n to reach server s, $hop(n, s) : N \times S \to \mathbb{N}$
$hops$: a matrix of hop distances between every node in N and every server in S, where $\{n_1, n_2, ...n_{|N|}\} = N$ and $\{s_1, s_2, ...s_{|S|}\} = S$. $hops =$

$$\begin{gathered} hops(n_1, s_1), hops(n_1, s_2), ...hops(n_1, s_{|S|}) \\ hops(n_2, s_1), hops(n_2, s_2), ...hops(n_2, s_{|S|}) \\ ... \\ hops(n_{|N|}, s_1), hops(n_{|N|}, s_2), ...hops(n_{|N|}, s_{|S|}) \end{gathered}$$

2.2.2 Communication metrics

$l(n)$: the number of transmissions undertaken by node n, $l(n) : N \to \mathbb{N}$
$L(c)$: the number of transmissions that occur in cluster c, $L(c) : C \to \mathbb{N}$

$$L(c) = \sum^{n \in N} l(n), \text{ with } cluster(n, c) = 1$$

L_t: the total number of transmissions in the network

$$L_t = \sum^{c \in C} L(c)$$

$\bar{L}$: the average communication load of all clusters

$$\bar{L} = \frac{L_t}{|C|}$$

$B(c)$: the load balance metric of each cluster is

$$B(c) : C \to \mathbb{Q}, B(c) = |L(c) - \bar{L}|$$

$q(n)$: the highest end-to-end latency suffered by node $n, q(n) : N \to \mathbb{N}$
$Q(c)$: the highest end-to-end latency suffered within the cluster $c, Q(c) : C \to \mathbb{N}$

2.2.3 GNN metrics

$x_n = [config(n), hops[n]]$
$X = [x_{n_1}, x_{n_2}, ...x_{n_{|N|}}]$, where $n_1, n_2, ...n_{|N|} \in N$

2.3 Formal description

Given the notations above, let there be the graph $G(N \cup S, E)$, where $N \cup S$ is the set of nodes and the set of server nodes in the network, and E is the set of links in the network. A communication link is formed between any two nodes if each one of them is within the transmission range of the other. Each node n in the network needs to send its own data every p seconds to a server node s. n can both receive messages and send messages. They send their own data and forward the data of other sensor nodes toward the destination. s just wait to receive data. For each s, a cluster c is defined. c is a group of multiple n and one s. During timeslot t, n can either send or receive data, but not both.

The goal of the load balancing approach is to minimize the difference between the load of each cluster and the average load between all clusters by changing the clustering configuration of individual nodes. More concisely, this translates to:

$$\begin{gathered} min_{config(n)} B(c), \\ \forall c \in C, \\ \forall n \in N \end{gathered}$$

This goal will be achieved by creating a custom unsupervised loss function to train a classifying GNN. The GNN will take the feature representation of the nodes and the graph representation of the network and output the new clustering configuration of the nodes.

Chapter 3

Technical background

3.1 Introduction to Internet of Things

3.1.1 IoT Systems

Internet of Things as a concept aims to create a connected world of devices. In simple terms, the idea is to be surrounded by a multitude of "things" which are connected to the internet and, by extension, to each other. The "things" are the end devices that can vary wildly, sensors, wearable devices, smart fridges, basically any electronic device with an embedded chip and Internet access.

As a growing market, IoT has been developing on all fronts tremendously. While the end devices get the biggest spotlight, it is good not to forget how many components are part of this vision. Communication protocols are constantly trying to outperform each other and solve the congestion issue that usually happens on the 2.4 GHz ISM band[18]. In this regard, a relatively recent shift was 5G, which promised extreme improvements over conventional protocols[19]. On the cloud side, there is a constant expansion of data centres, storage technologies, security measures, etcetera. Devices themselves have become cheaper and more power efficient. This has allowed for the commodification of the concept, as well as an increase in the hobbyist market for people that want total control over their data. Like with many other domains, machine learning has wiggled itself into this complicated mesh of things with the promise of better performance. All of these parts come together to form the ever-changing interconnected environment that is IoT.

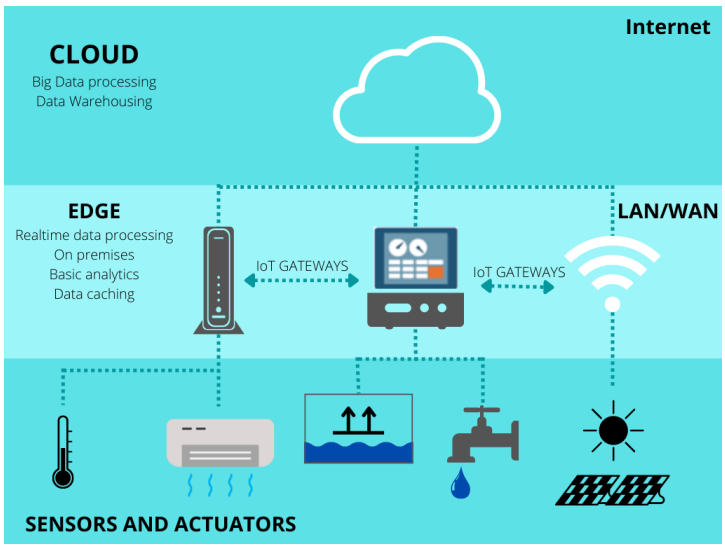

Figure 3.1: The IoT infrastructure

In broad terms, the IoT infrastructure can be split into three: the end devices, the edge gateway, and the cloud infrastructure. The end devices are what most people think of when they hear IoT, things like thermostats, smart light bulbs, etcetera. These devices can generally be split into sensors and actuators. Edge gateways provide a bridge between the devices and the cloud. The roles these can take can vary, from simple data aggregation to being edge servers that also do some computation. The cloud's main purpose is to store and process the data. An overview representation of IoT infrastructure can be seen in Figure 3.1.

The notion of "intelligence" or "smartness" is given by the ability of entities to react to given inputs and events. Any of the previously mentioned components can be intelligent in one way or another. The level of intelligence can vary depending on several factors, but one of the most important is the nature of the application that is serviced. Elements like the time sensitivity, amount of data, nature of the data, and scope of interaction also play a part. End devices can take action immediately but have minimal computation power and level of interaction. The cloud has immense levels of storage and computation but is the slowest to respond to events. Edge gateways are a compromise between these two. Based on this, decisions can be taken at different levels in the pipeline depending on scope and time sensitivity.

3.1.2 Wireless Sensor Networks

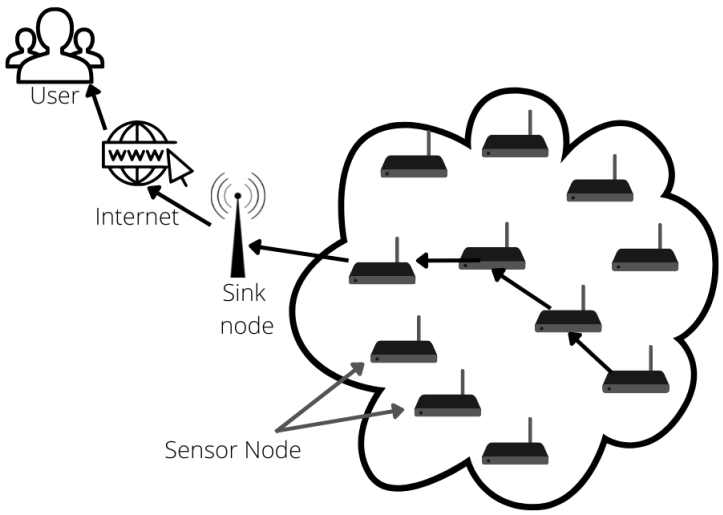

Figure 3.2: Wireless sensor network

Wireless Sensor Networks (WSN) are a well-known concept in the networking field. WSN are a formation of multiple simple devices collaborating towards a single goal. Each device usually has just a sensor, controller, battery and transmitter. They collect data from the physical environment and aggregate it at the edge gateway to be further processed. Alternatively, the notion of Wireless Sensor Actuator Networks (WSAN) exists. As the name implies, the main difference is that within the network, some or all of the nodes have an actuator. While this difference may seem small, it changes the communication paradigm completely. The bulk of communication in WSN happens upstream, from the end device to the edge gateway. All of this communication transmits application specific data. There is also a much smaller portion of communication downstream, from gateway to device, usually transmitting routing and scheduling information. With WSAN, the importance of upstream and downstream is more balanced since a lot of communication happens from gateway to actuator devices. In practice, however, these two terms will be used

interchangeably, mostly calling WSN any network regardless of whether it has actuators or not.

In terms of application types, there is no clear bound to what WSN can be used for, and their broad concepts can be adapted to fit a variety of real-world application scenarios. As such, there is a wide degree of diversity in both requirements to be fulfilled and technical methods to be employed in their implementation. A general limitation, given by the nature of the deployment, is that the end devices are most likely battery powered. This created the innate desire for good energy efficiency, which would ensure a longer network lifetime. Better energy efficiency can be achieved through multiple avenues and at different levels in the pipeline. While the power concerns are important, they may not play the primary role in the given application scenarios. Several other goals may take centre stage. Security will be the highest regarded goal in fields that deal with sensitive or classified data, like the military sector. Speed or latency might be the primary target in real-time systems such as factory automation. Reliability, availability, and jitter can serve as goals in designing the network, depending on the application at hand.

Typically, in a WSN, there are two roles to be defined, sources and sinks. Sources are the producers and senders of data. Sinks are the receivers of data and sometimes, depending on the use case, also senders of control data. The number of sources generally is higher than the number of sinks. This has to do mainly with the fact that most, if not all, sensor nodes are going to be sources. Since sinks are the points where data is aggregated, the requirements for this role are considerably higher, both in terms of computational power and power supply. The sink role usually would be undertaken by the edge gateway.

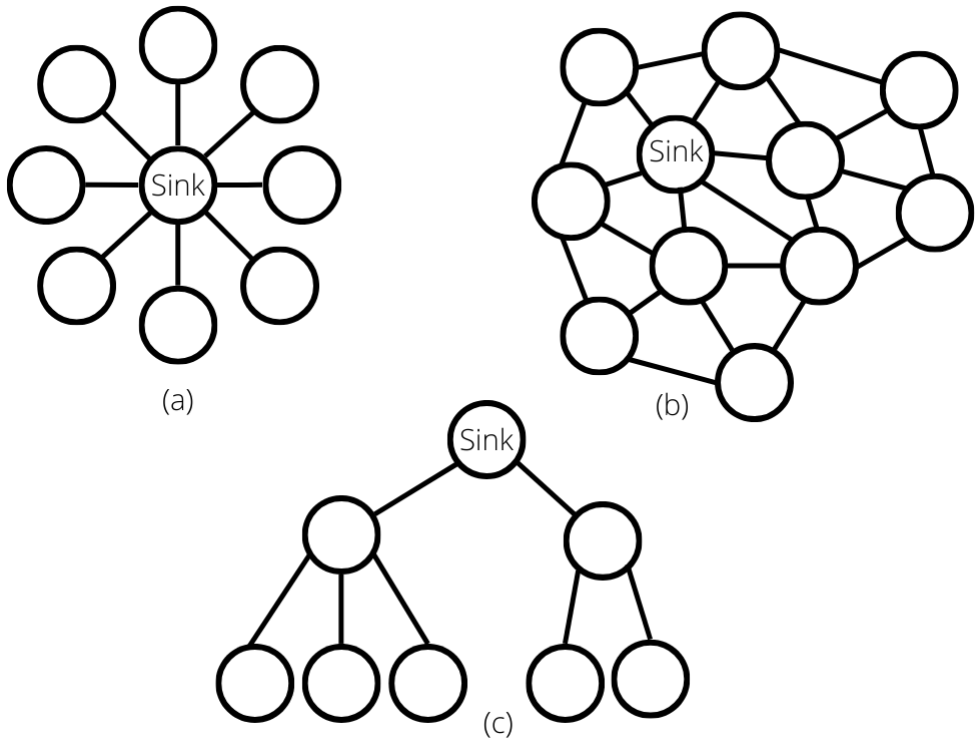

Figure 3.3: Different network topologies: a) Star b) Mesh c) Tree

WSN topologies can classified into multiple distinct shapes: point-to-point, tree, star, mesh, etcetera. In Figure 3.3, multiple WSN topologies are visualized. Depending on the nature of the application, each of these topologies has its advantages and disadvantages. Alternatively, some applications can just run under certain topologies, which might be a limitation imposed by the environment, communication protocol, etcetera.

3.1.3 Mesh Networks

Mesh networks are non-hierarchical in nature, which means there is no inherent structure to the network like there is with a star or tree network. With star and tree topologies, the gateway is generally connected to a small set of devices that either have no connection between themselves or have very strict patterns. In mesh networks, devices and gateways connect to as many neighbouring devices as possible to form a strongly connected network. Devices and gateways are seen as nodes in the mesh that work together to further the application goal. This structure gives independence from any one particular node, making mesh networks node fault tolerant. Additionally, it is not uncommon to see alternative route retransmission in case of delivery failure. Mesh networks can self-configure and self-organize. This ability to self-configure gives mesh networks the option to respond to node failure by creating alternative routes, if possible. There still can be critical network failure because of nodes disconnecting, depending on which nodes disconnect and the topology of the network. However, it is generally more fault-tolerant and, thus, less costly when it comes to maintenance. The drawback is the added complexity of the routing and communication protocols that have to operate on mesh networks. Additionally, the increased connectivity can lead to increased congestion and make it harder to create schedules for node communication, which leads to more retransmission and power usage.

3.2 Introduction to Graph Neural Networks

3.2.1 Machine learning models

IoT is notorious for the amount of data it produces. It clearly contributed to this era of big data, where billions of connected devices generate petabytes of data [20]. It is easy to understand society's desperate search for an efficient way of analyzing and interpreting these colossal amounts of information. In this regard, Machine Learning (ML) has quickly become a favourite approach to tackling this issue. At its core, ML is about building mathematical models that need to be tuned for specific tasks. The models do not need to be specifically programmed to find certain patterns; they infer them based on the data given to them. This makes finding and curating the input data a task as important, if not more important, than creating the model itself. The correctness and accuracy of these predictions are highly dependent on the given input data.

ML models are usually composed of layers, which are then composed of neurons. Neurons are essentially nodes that represent a certain entity. The relations between these neurons are mathematical formulas with trainable weights attached to them. These weights are constantly adjusted during the training process to obtain better results. Between the neuron layers, there exist activation layers. The role of these layers is to emulate synapses firing, selecting which and how the output of the previous layer will be fed to the next.

3.2.2 Types of learning

Generally speaking, ML can be split into four distinct categories, supervised learning, semi-supervised learning, unsupervised learning and reinforcement learning.

Supervised learning is possibly the most widely used and usually has the highest prediction accuracy [21]. This comes at a steep cost; these models need to be fed labelled data. Labelled data are data sets for which the correct answer is known. For example, in an image classification task, a specific image is known to represent a certain thing, like a dog. This information is known and helps in training the model. In its learning process, a supervised model will compare their predicted result with the ground truth and try to adjust its weights to match the real-world results better. Afterwards, the model will be tested. The model is given new data it has not seen before and computes how well the predicted results compare to the ground truth.

A semi-supervised model is very similar to a supervised model. The main difference is that not all its data is labelled. Usually, the training set is labelled in order to train the model for the problem at hand. Since the testing phase does not have labelled data, for asserting its performance,

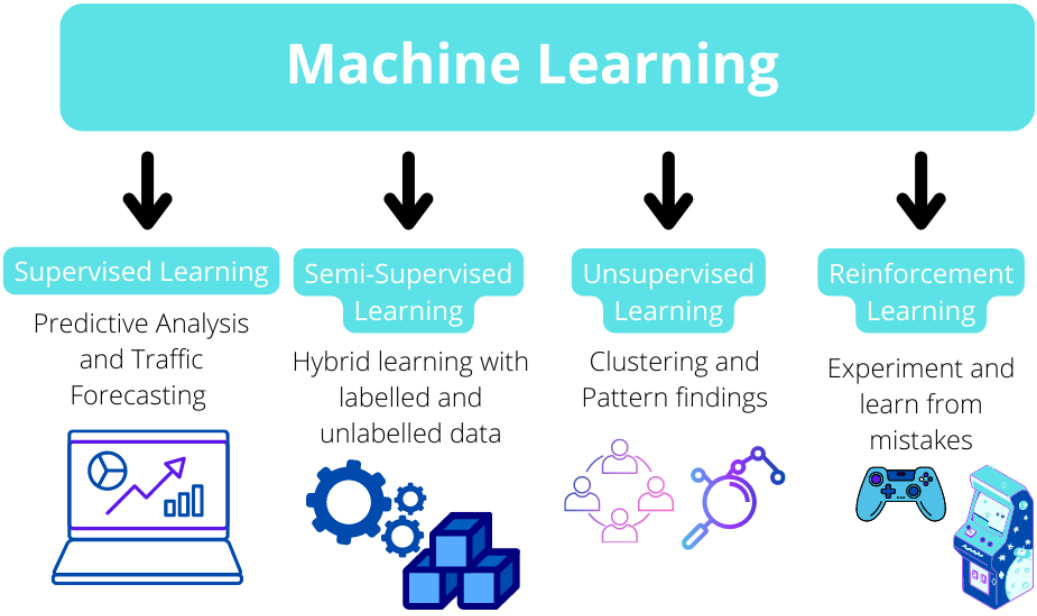

Figure 3.4: Types of Machine Learning

new metrics need to be created. These metrics are usually tailor-made for the given task. The method is advantageous since it requires considerably less labelled data than a fully supervised approach. Labelled data is very time consuming to obtain since someone needs to manually go and assign labels to all the data points in the set. This procedure is usually the most consuming when creating a new implementation [22]. Additionally, it avoids the issue of creating a custom loss function since it can just use a form of entropy to compare to the ground truth.

Unsupervised approaches, as the name implies, require no labelled data. This usually requires the creation of a custom loss function as well as testing evaluation metrics. The loss function, also called the objective function, basically measures how well the model performs and tries to adapt the given task into a usable form for the model. The challenge moves from assembling a proper data set to creating a useful loss function in this approach. While not as time-consuming, this process is much harder to achieve since the designed function might not have the expected outcome. This speaks of the limitations of ML in general. Creating a loss function that performs as intended and provides good performance is hard to achieve. It does not help that, since this approach does not have labelled data, it usually provides a much more general approach that performs worse than its supervised siblings.

Reinforcement learning is conceptually different from the rest of the approaches. It is trained through an action-reward system. With reinforcement learning, there is an agent and an environment. The agent takes action in the environment and gets a reward based on how good that action was. The agent tries to achieve the maximum cumulative reward through this continuous interaction. The environment should be able to respond to the agent's action and assert how good or bad this action was. The main challenge with reinforcement learning is the trade-off between exploration and exploitation. Exploitation means choosing the highest reward action based on previous experiences. Exploration means choosing a yet unknown action to see what reward is given. There is a delicate balance between these two. Too much exploitation can lead the learning to be stuck in local maxima of the problem, while too much exploration can mean the model is not learning correctly [23].

3.2.3 Use of graph structures

GNN are a type of ML model. The main difference does not come from the learning type used but from the type of data structure used as input. As the name would imply, GNN models use graphs as input. Graphs are a type of data structure that is composed of nodes and edges. Nodes represent the entities of the modelled data, be it people, vehicles, IoT devices, etcetera. Edges represent relationships between nodes, and these can be almost anything. For example, from who knows who, how related two nodes are, and if two devices communicate with each other.

The reason graphs are important is because they can model non-Euclidian problems. In normal ML, the input is usually sequential, be it characters from text or pixels of an image. These types of inputs usually describe simple to understand relations. Each element is influenced by adjacent elements in an Euclidian space. For example, one pixel in an image will be directly influenced by the adjacent pixels. Sequential approach might not be optimal for some problems, like social networks, for example, where one user can have a varying number of different relations with multiple users simultaneously.

3.2.4 Understanding the process

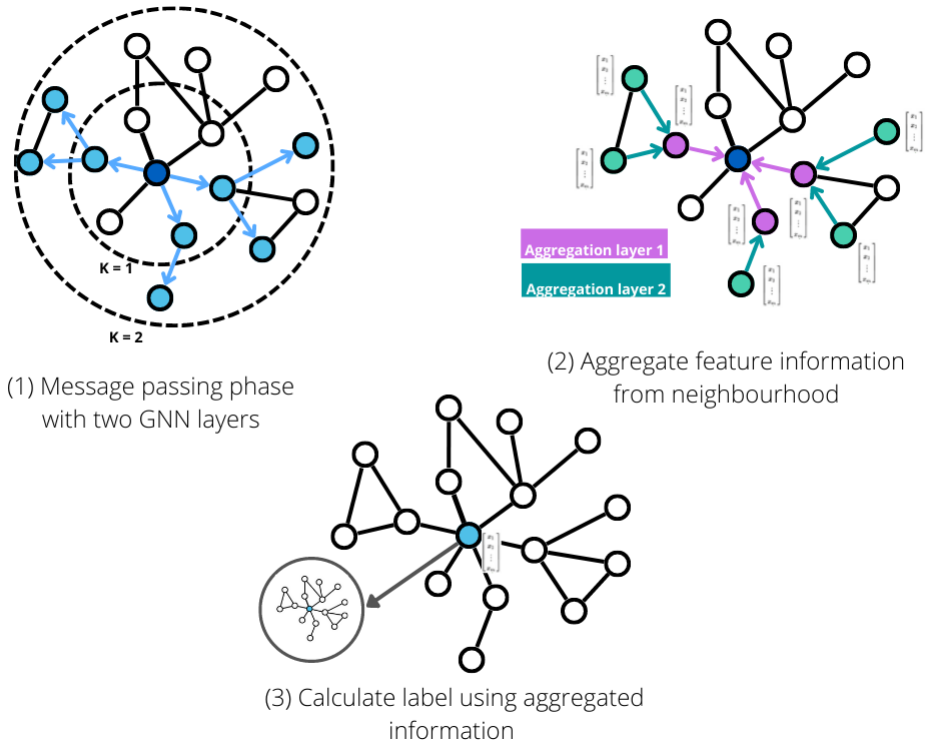

Figure 3.5: Process of obtaining node embeddings

The main concept to understand about GNN is node embeddings. Each node in the graph starts with a feature set, which becomes its initial embedding. This can be comprised of any relevant information for that node in particular and is dependent on the context of the data.

$$h_v^0 = X_v$$

Consider the entire GNN as a set of sequential states. The initial state of each node is its initial feature set. Each layer of the GNN is the transition step between two states. This transition step can be represented as a message passing phase, an aggregation phase and a neural network phase.

During the message passing phase, each node gathers the current node embedding of its neighbours. In the aggregation phase, all the neighbour's node embedding and the node's current embedding are combined according to the aggregation function. A visual representation of this can be seen in Figure 3.5. The aggregation function can take many shapes, from a simple add or mean to a whole neural network of itself, for example LSTM. In the neural network phase, the combined embeddings are put through a linear layer to obtain the new node embedding. This gives GNN the trainable functionality. This is a generalization of the process. In practice, there can be variations, but these three phases are usually maintained. A very generalized formula of the process can take the form:

$$h_v^k = \sigma(W_{1-k} * aggr(h_u^{k-1}) + W_{2-k} * h_v^{k-1})$$

with σ being the layer activation function; W_{1-k} and W_{2-k} being the trainable weights of the layer; $aggr()$ is the chosen aggregation function; u represents all the neighbours of v or $u \in \mathbf{N}(v)$; k = 1, 2,..., K-1

After the GNN is applied on the input data, the resulting output of that node is called its final node embedding. Each node embedding is the learned representation of that node and its relations. The model is trained in accordance with its defined loss function.

$$Z_v = h_v^K$$

From a design perspective, each layer increases the number of nodes considered for the final node embedding by increasing the degree of neighbours to be searched. This has a large impact on the results. While more layers capture the details of a wider area of nodes and relations, this can lead to gross generalizations of information, where the initial data is diluted too much. Additionally, this increases the computational requirements of the process significantly. The decision on the number of layers is a trade-off between the importance of initial data and the number of relations to be considered.

3.2.5 Types of GNN problems

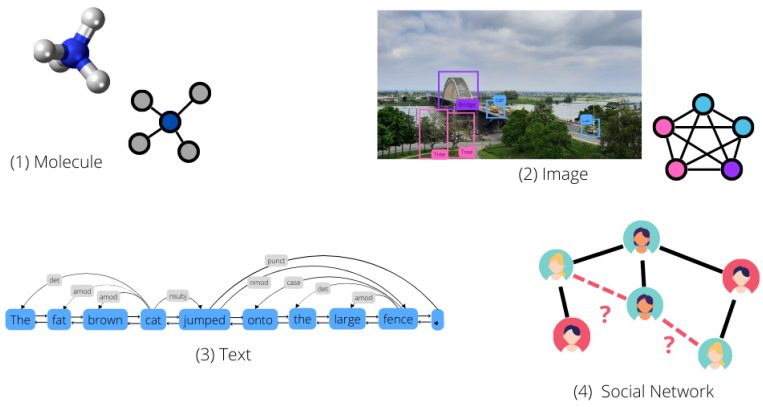

Figure 3.6: Example application scenarios for GNN

The types of problems GNN can approach are varied and very context dependent. The following are just some of the most common types of problems GNN is used on.

1. **Node classification:** the problem is to assign a label to nodes. GNN has an inherent advantage here because it can offer predictions based on both the node's feature set and its relations with other nodes. Examples of use cases include citation networks [24], where the goal is to identify the topic of an article based on its abstract and its citation list, which serves as a relation to other articles. This task is usually done in a supervised or semi-supervised manner.

2. **Graph classification:** is the problem of assigning a label to an entire graph. For this problem, patterns need to be identified in the structure of the graph in order to classify it. Usually, for this type, some form of pooling method is used to summarize the features and relations of entire graphs to make things more manageable. This problem is most common in the field of chemistry, where molecules and other compounds can be represented as graphs and then they need to be classified [25].

3. **Link prediction:** this problem deals with the relations between entities. The GNN needs to understand the relationships between the nodes on the graph in order to be able to make predictions on possible connections between entities [26]. This problem is common in recommender systems where possible connections need to be made for users based on their previous history, be it shopping or social media friends.

Chapter 4

Related works

4.1 Unsupervised GNN and Representation learning

The model proposed by Kipf et al. [27] has been instrumental in the further development of GNN. It provided a simple spectral approach to the problem of message passing. GCN provides a simple solution for creating node embeddings. It can be used as a building block to further develop a multitude of different implementations [28] [29] [30] [31] [32]. In our case, it could be useful for creating the initial node embeddings on which we apply a different clustering technique or be used in a more direct combination of the embedding an clustering process. While other models are also technically Graph Convolutional Networks (GCN), the implementation by Kipf has become synonymous with the term.

The idea behind a GCN layer is to use a modified adjacency matrix in the process of message passing. This is done as a simplified version of spectral convolutions tailored for use in deep neural networks.

The following formulas for GCN are given by Kipf et al..[27]

First, the identity matrix is added to the adjacency matrix of the graph.

$$A' = A + I, \text{ where } I \text{ is the identity matrix.}$$

Next the degree matrix is computed such that:

$$D'_{nn} = \sum^{n' \in N} A'_{nn'}$$

With these, we can compute what the paper calls a renormalized Laplacian.

$$A\text{"} = D'^{-\frac{1}{2}} * A' * D'^{-\frac{1}{2}}$$

A GCN layer then takes this new matrix and uses it in the message passing phase.

$$Z = A\text{"} * X * W$$

where Z is the output of the layer, X is the input feature set of the graph, and W is the matrix of trainable weights.

GraphSAGE [33] was developed as a response to the transductive GNN architecture. It proposes an inductive GNN architecture that should work better for unseen nodes. This is due to the ability to better generalize relations based on neighbourhood structure. Additionally, it supports the use of multiple aggregation functions in its design. The spatial nature of this model, combined with the modularity when it comes to aggregation functions, make it a very appealing building block in our design for the purpose of creating the node embeddings.

The first difference between GraphSAGE and GCN is how they operate on the graph. GraphSAGE needs to compute the new state of each individual node. Additionally, GraphSAGE is a spatial convolutional model where the embeddings are based on the direct neighbours of each node.

GraphSAGE formulas are stated bellow.[33]
Assume the initial state of each node is its feature.

$$h_v^0 = X_v$$

First the aggregated state of the neighbours is calculated

$$h_{\mathbf{N}(v)}^k = aggr(h_u^{k-1})$$

with $aggr()$ is the chosen aggregation function; u represents all the neighbours of v or $u \in \mathbf{N}(v)$; k = 1, 2,..., K-1 and represents the number of the GraphSAGE layer
The neighbourhood state is then used for the state of the current node.

$$h_v^k = \sigma(W_k * concat(h_v^{k-1}, h_{\mathbf{N}(v)}^k)$$

with σ being the layer activation function; W_k is the trainable weights of each layer; $concat()$ is the concatenation function.
This leads to the final embedding of a node being:

$$Z_v = h_v^K$$

Deep Graph Infomax (DGI)[34] is a general method for unsupervised learning of node representations inside graph data structures. The main functioning principle is maximizing mutual information between neighbourhood level information and graph summaries. The feature of each node in this context would become the representation of the neighbourhood centered around the respective node. Both the neighbourhood representation and graph summary are computed through graph convolutional architectures. DGI would serve the role of an encoder and need another mechanism for clustering.

GRACE [35] offers a novel framework for unsupervised graph representation learning by utilizing a contrastive objective at the node level, which is motivated by recent success of contrastive approaches. To achieve this, it creates two corrupted views of the network by removing edges and hiding node features. The idea behind this is the maximize the agreement of node embeddings between the two corrupted network views.

The representation learning methods described earlier are mainly methods of capturing within the node embeddings the whole context of the neighbourhoods they are the center of. Each is good in their own right, but not all of them are equally valuable for the purposes of our project. DGI and GRACE add a high amount of computational complexity that is not needed. The main problem of our project is not the correctness of the embeddings, but how to manipulate the embeddings to better represent a load balanced configuration of the network. The decision has to be made between GCN and GraphSAGE. In the end, GraphSAGE was chosen because of the reasons enumerated in Section 5.1.

4.2 Baseline network clustering approaches

Voronoi [36] is a widely known and used naive solution for clustering problems. Its simplicity is what makes it so useful in a wide variety of cases. The concepts behind Voronoi are simple to understand. For the purposes of IoT networks, by applying Voronoi clustering, each node in the network will be assigned a cluster based on which server is the closest in proximity. For our purposes, this proximity will be the number of communication hops it would take a message to reach the server.

The clustering solution Tsitsulin et al. [37] stands out for multiple reasons. The first one is that it uses GNN in order to cluster graphs. The second and more important aspect is that it can do so in an unsupervised manner. For our purposes, this has proven invaluable. In the field of GNNs, what most implementations use is a semi-supervised solution for clustering. Clustering, in this sense, is actually a node classification task. However, in the real world, there is not always prerecorded usable data for a GNN to learn. This implementation uses modularity calculation

for the purposes of clustering graphs. Modularity is the measurement of how well connected a clustered network is. The more intra-cluster edges and the fewer inter-cluster edged are, the higher the modularity score. There is one caveat, the Tsitsulin implementation is designed to be used on graphs. While networks are usually represented as graph structures, they have a different set of requirements than normal graphs. One important aspect is that in a network, distances do actually have a meaning. Unaltered, the Tsitsulin implementation is not usable for networks; it would output unusable results. For this purpose, an alteration had to be made to the original design. The change was the introduction of a hop distance component in the loss function of the model. This new element would ensure the internal cohesion of the network clusters.

4.3 GNN based graph clustering

CAGNN [38] is a cluster-aware graph neural network model for self-supervised graph representation learning. Their approach takes three steps. In the first step, node embeddings are generated through graph convolutions. In step two, the model trains in a self-supervised manner by iteratively optimizing the parameters based on predicting cluster labels and updating the clustering for the nodes. They formulate cross-entropy minimization as an optimal transport problem, to avoid degradation of results. The final step is weakening links between clusters and strengthening links within the same cluster. The last step is the reason why this model is not applicable for our context. It goes on the assumption that better modularity is a good thing. Modularity is a metric that measures how well a graph can be divided into subgraphs. In our network context, links are not relations, but show possible communication. Additionally, the self-supervised nature of this approach would not fit well in our network environment due to the fact that the server nodes are fixed points of interest around which the clusters need to revolve.

MinCutPool [39] is a graph clustering approach that tries to circumvent the limiations of Spectral Clustering. It could provide the basis for the clustering part of the problem. It trains a GNN to minimize a normalized version of the minCUT problem[40]. However, the real focus of the paper is on introducing a pooling layer able to summarize the features of all the nodes in the graph into a graph embedding. The main issues with MinCutPool are that the regularization used in the loss function is to strict and hinders the model in learning properly and that the approach cannot handle uneven connectivity of the graph.

SEComm[41] is a self-supervised community detection model that attempts to circumvent the usual restriction of semi-supervised approaches. It uses GCN[27] to encode the node embeddings and DGI[34] to learn the node representations in a self-supervised way. An intermediary step is to learn to predict the similarity matrix between the aforementioned node representations. With the predicted similarity matrix, an MLP model is trained to obtain the actual clustering of the network. The process described in this paper is very computationally taxing, because of all the models that need to be trained. The main reason why this approach would not work for our problem is that the principle of self expressiveness[42], used in the predictions of the similarity matrix, works for nodes with a high dimensional node embedding, from which it can draw information. Nodes in our context do not have the necessary information for this approach to work.

Khanfor et al.[43] proposes a method for clustering large scale Social Internet of Things (SIoT). This paper provides a very interesting and rare perspective into how GNN would be applied in the IoT network environment. There are two components to this method. The first component is embedding the characteristics of each device and its social relationships using a semi-supervised GNN. The second component is forming the clusters using one of two approaches, either k-means[44] or DBSCAN[45]. This approach is unsuitable because of two main reasons. The first reason is that node embedding is done is a semi-supervised manner while we have an unsupervised task. The second reason is that the clusters formed end up being communities of similar values instead of clusters for communication purposes around a fixed server node.

4.4 Dynamic neural network models

EvolveGCN [46] is a model that intertwines a Graph Convolutional Network (GCN) and a Recurrent Neural Network (RNN) in order to capture the time dynamic events that happen in the graph structure. This model can be a powerful tool to build on top of and adapt depending on the application needed. The general idea is that the RNN would be used to update the parameters of the GCN at each time step. They actually propose two implementations, one using a Gated Recurrent Unit (GRU) as the RNN and one using Long Short Term Memory (LSTM) as the RNN. They claim that the GRU version is more suited in cases where the node features are more informative and the LSTM version is more suited for cases where the graph structure is more important.

TRNNGCN [47] is a model combining Recurrent Neural Networks with Graph Convolutional Networks in order to solve the problem of clustering graphs with dynamic node-cluster membership. This model could serve as a starting point for solving the load balancing problem on IoT networks. It propose a dynamic stochastic block unit to capture the time dynamic behaviour of nodes. Their model clusters the graph by predicting the decay rate of cluster membership for nodes.

There are a few reasons why the aforementioned papers are not usable. First aspect is that the implementations are meant for semi-supervised tasks. Even if they could be adapted for unsupervised tasks, they are not suited for the context of the problem we are trying to solve. Links in the network structure do not represent relations between nodes, but the possibility of communication. Information about the nature or pattern of dynamic events cannot be gained from studying the links. From the perspective of the GNN, two time snapshots of the network can be interpreted as two distinct networks.

Liu et al.[48] present a very interesting Deep Reinforcement Learning(DRL) model for load balance aware clustering in an edge server scenario. They use a Dueling Double Deep Q-Learning Network for their prediction purposes. The goal is to find a load balanced clustering solution for dynamic networks. The scenario and problem on which their solution is applied are very similar to the ones for this paper. However, there are a few crucial differences in approach. The first has to do with how dynamic events are handled. In their paper, nodes are assumed to follow predictable movement patterns that are able to be understood by a LSTM architecture. Additionally, the movement of nodes is observed in motion. This movement is also important for the purposes of the neural network. More precisely, the direction of the movement is used as input for the DRL model. Static nodes in the network periodically scan for dynamic events. In the case of our approach, only the end result of the movement is observed, not the movement itself. These differences, coupled with the added model complexity of including DRL with GNN have made us exclude this paper from influencing our solution.

4.5 GNN frameworks

Pytorch Geometric [49] is a library for use with Pytorch, which adds functionality for creating, training and testing GNN. It has proven to be an invaluable tool in expanding the interest in the field of GNN, as well as lowering the barrier to entry for research on this topic. It provides already implemented models from several GNN papers, as well as multiple data sets for GNN training and testing. This library, alongside its implementation of the previously mentioned papers, has been used during the development of this thesis.

Chapter 5

System design

5.1 The GNN structure

The project's goal is to learn a function that maps each node to a clusters. The clustering is based on the feature set of that particular node and the adjacency matrix of the network. This clustering aims to achieve minimal $B(c)$, $\forall c \in C$. This needs to be done without any ground truth clustering information.

To achieve this mapping, we create a model containing two GraphSAGE layers, explained in section 4.1, with addition as the aggregation function. As such, given the graph G, the output mapping of the graph is equal to:

$$Z = f(X, A) = Softmax(GraphSAGE_2(Dropout(ReLU(GraphSAGE_1(X, A))), A)$$

GraphSAGE was chosen for the layer structure because of the spatial nature of the approach. In our use case, spatial information is of high importance. The goal of the implementation is to create disjoint clusters that strive to be balanced from the perspective of load. To maintain cluster unity, the goal is for nodes at the extremity of clusters to change cluster. The loss function plays a very important role here, and it is discussed in the next section. Another component that plays a role is the neighbourhood of the node. The neighbourhoods of the nodes that are at the extremity of the cluster will naturally have more nodes from different clusters. With a spatial GNN approach, this neighbourhood will be the one to directly influence the embedding of the target node. In both theory and practice, this leads to nodes that are more willing to change clusters when comparing GraphSAGE and GCN. This is a hard difference to notice in practice. It can generally be seen through slightly better convergence of GraphSAGE, as seen in Figures 5.1 and 5.2. These experiments were run on the random network dataset, with the best setup described in Table 6.1

The reason two GNN layers are used is that two GNN layers capture enough of the structure of the neighbourhood without diluting too much of the initial features of the node. Especially in the case of networks, where there is a clear delimitation of the clusters, too many layers would mean increasing the likelihood that nodes become influenced by other nodes from different clusters.

The input of the model is comprised of two elements. The first is the graph's adjacency matrix in the form of an edge list. The second is the feature set of all the nodes. The feature of each node should contain as much information as possible for that particular node in the hopes of making the feature as unique as possible. Choosing the node feature is an essential step, especially for node classification models, since identical or very similar features would lead to comparable label attribution, which might not be necessarily desirable. For this particular case, the feature of each node is comprised of its clustering label, as well as the hop distance list to all the servers, discussed more in depth in Section 5.4. In practice, this means the input dimension is twice the size of the number of server nodes, $2 * |S|$. By the definition of the network, this is all the usable identifiable information of each node.

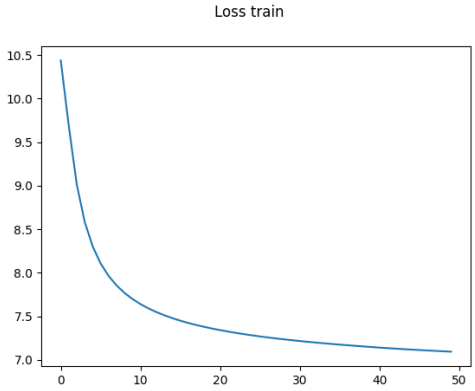

Figure 5.1: Loss curve with GCN layers

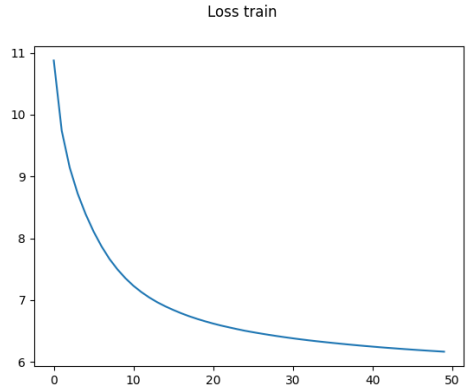

Figure 5.2: Loss curve with GraphSAGE layers

$$x_n = [config(n), hops[n]]$$
$$X = [x_{n_1}, x_{n_2}, ...x_{n_{|N|}}], \text{ where } n_1, n_2, ...n_{|N|} \in N$$

The number of hidden neurons is discussed more in depth in Section 6.4.4. There was one choice that was made here. Normally, the number of hidden neurons is set to a fixed value, usual power of 2. Because of structural reasons, a GNN model can only be used on networks with the number of servers it was designed for. This creates a strong coupling between model and number of server nodes. Choosing a fixed value as the number of hidden neurons for all the networks would be problematic. It would give varying results based on the number of servers the model is used for. To stabilize this behaviour, the decision was made to link the number of hidden neurons to the number of server nodes. This would give the same level of expressiveness to all models regardless of the number of server nodes.

The number of output neurons is equal to the number of clusters, which is also equal to the number of server nodes. Our GNN model processes nodes individually. After all nodes in the network are processed, the output is combined into the output matrix Z. In the output matrix Z each row Z_n represents a soft cluster membership vector of size $|C|$ of node n, such that each element Z_{nc} represent the probability of node n to be in the cluster c. The final output matrix Z is then of shape $|N| * |C|$.

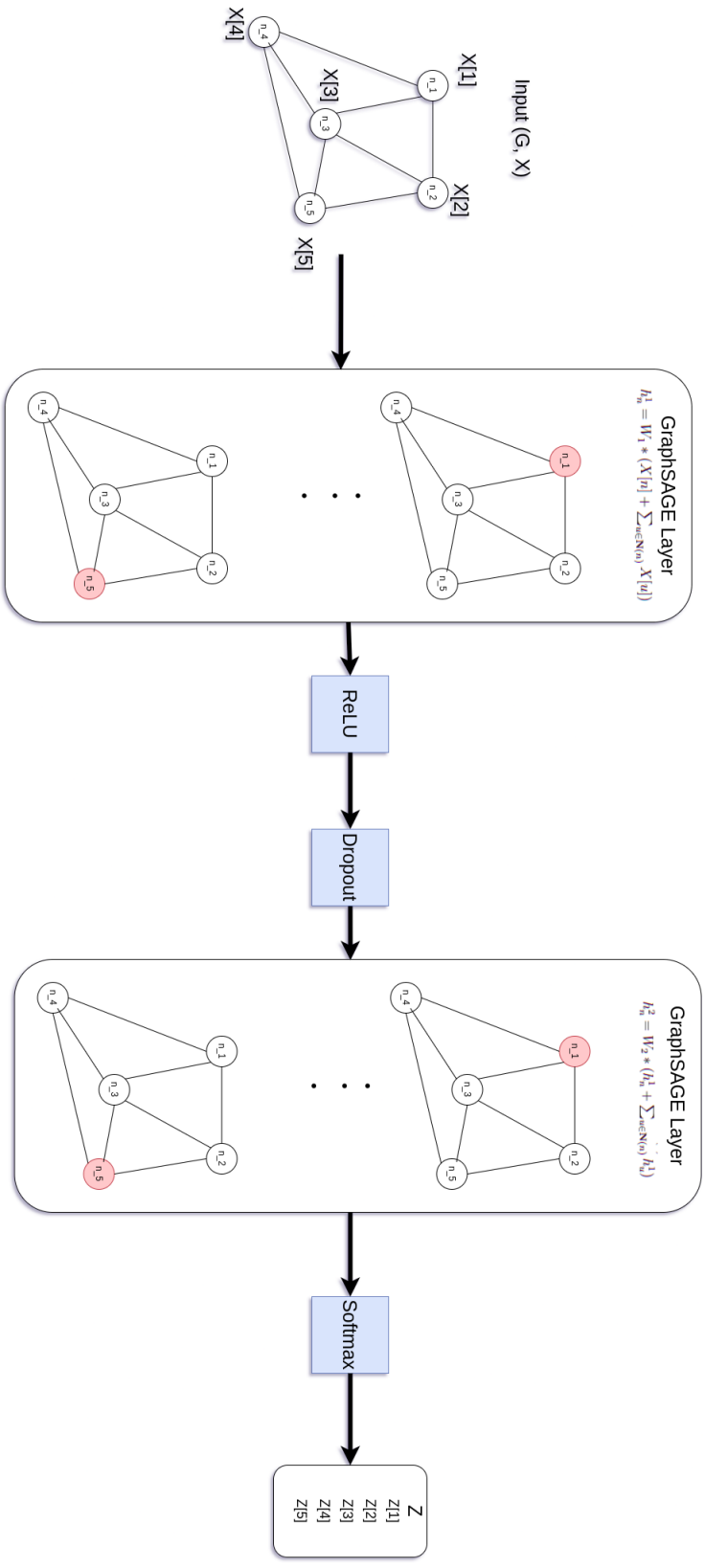

Figure 5.3: Architecture of GNN model

5.2 Objective function

The main objective of this approach is to load balance the wireless mesh network. This means minimizing $B(c), \forall c \in C$. To relate this to Z, the output of the GNN, a network-wide version of $B(c)$ needs to be created that takes into account the soft clustering. This loss function takes the form:

$$min_{W_0,W_1}\mathcal{L}_{LB} = ||(Z \times l(N)^T) - \bar{L}||_F$$

where W_0 and W_1 are the weight matrices of each of the GraphSAGE layers; Z is the output of the GNN; $l(N)$ is the load vector of all nodes in the mesh network; $\bar{L}$ is the mean load between clusters; $||\ ||_F$ is the Frobenius normalization of the matrix

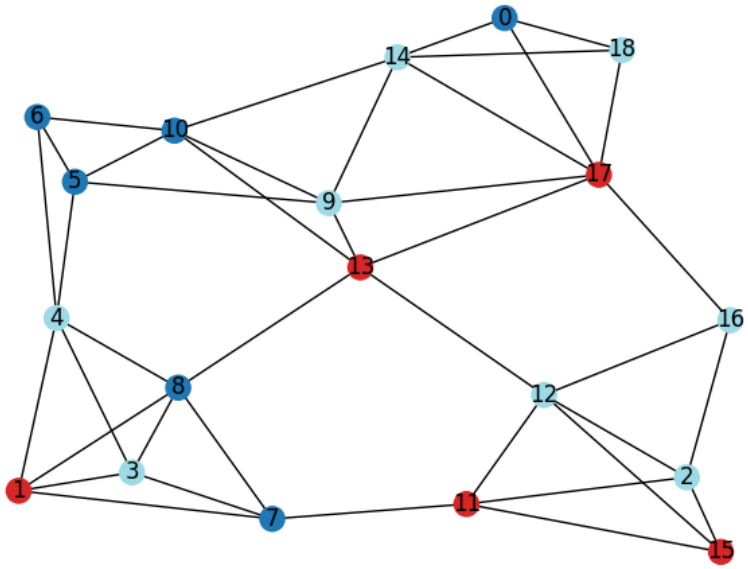

Figure 5.4: Example clustering just with load balance

However, focusing on just load balancing leads to unusable results in the context of a wireless mesh network. It would result in nodes being assigned to server nodes such that the clustering calculation would be minimal, without any regard for their position. In mesh networks, the node's position is highly important, and it is what dictates the edges it will have with other nodes. This means that besides the clusters needing to be disjoint, the position of nodes plays a huge role in the clustering process. An element that takes into account the position of the node needs to be added to the loss function. The best choice would be to use hop distance to the servers. The geographical position is hard to calculate and maintain and is somewhat irrelevant to this problem. Hop distance bounds the position of each node concerning the server nodes, which are the focal points of the mesh network. Additionally, this hop distance loss needs to be significantly stronger in comparison to the load loss. The desired outcome is to have clusters around the server nodes that exchange nodes situated at the extremity of the clusters. This would ensure cluster unity while attempting to achieve load balance by switching peripheral nodes between clusters. To this end, a linear relation for the hop distances does not have a steep enough curve to create the desired outcome. The desired outcome is for nodes further away from the servers to be substantially more willing to change clusters than nodes closes to the servers. This has led to the use of an exponential relation for creating a steeper curve. The decision was made that two was strong enough of an exponent because of the added effect the neighbourhood of the node has. For a node close to a server node, it is more likely to be surrounded by other nodes with the same clustering, making a

positive reinforcement loop. Nodes at the edge of a cluster are more likely to have nodes in their neighbourhood that are part of multiple clusters, thus already weakening the relation the current cluster.

$$\mathcal{L}_{HD} = \frac{\sum_{c=1}^{|C|} ||Z_c \times hops_c^2||_F}{|C|}$$

With this being considered, the final objective combines the previously mentioned functions. $min_{W_0,W_1}\mathcal{L}_{total} = \mathcal{L}_{HD} + \beta * \mathcal{L}_{LB}$

5.3 Creating the data and the simulator

The decision was made to use purely synthetic data for the purposes of the experiments. There is a multitude of reasons for doing this. Firstly, there was a need for a high number of IoT devices in order to form the networks. Secondly, there was also the need for devices fit to serve the role of edge servers for the network. All these devices would occupy a considerable amount of space, which would need to be reserved for the whole duration of the experiments, which would be months. Additionally, having a real-world testbed would open the experiments to outside interference, such as general building Wi-Fi. It can be argued that this would be beneficial since it would properly represent real-world conditions. However, it would also introduce an additional layer of randomness that cannot be controlled. The worst factor, however, is that all the devices would need to be repositioned to create each network. Given the fact that, during the time of this project, the number of networks used was more than a few thousand, it can be easy to understand how using a real-life testbed would quickly become out of hand.

Another option would be to find an already existing dataset that could be used for the experiments. However, using the reasoning from the previous paragraph, it was not possible to find a dataset that was big enough and also fulfilled our requirements for the experiments. With these options being unfeasible, the choice was made to use synthetic data.

With creating the data ourselves, there are two aspects to consider. The first aspect is creating the networks themselves. The second aspect is the network simulation environment. Both parts are very important in the overall process. While the network topology presents the particularities of the network and the adjacency matrix, the network simulation environment is the part that simulates the use of the network and calculates the statistics of the network.

For the purpose of creating a network, the network is defined by four aspects: length (in number of nodes), width (in number of nodes), number of server nodes and type of topology. The type of topology is what defines the datasets used in the experiments and is discussed in detail in Section 6.2. How the datasets are split is also discussed in that section for each dataset individually. The total number of nodes in the network is given by the formula $length * width + servers$. We chose to do it as such because the length and width define a 2D canvas on which nodes can be scattered. Additionally, the choice was made for server nodes to be added on top of the normal nodes. This decision was made to streamline the process and make the server nodes more intuitive. The server nodes will always be the first $|S|$ nodes both in code and in diagrams. This makes calculations easier and diagrams easier to read. The positions of both normal nodes and server nodes are highly dependent on the type of topology of the network. Generally, it can be one of three ways: fully random within the space of the canvas, random within a predefined area for that node or fixed. When nodes are initially scattered, there must be a minimum distance between nodes. This distance is a predefined value made to ensure that nodes do not overlap. While this is generally true, there are cases when this rule cannot be respected or is intentionally not followed. For example, trying to place a node in an area with a high number of nodes would be an impossible process if the minimum distance were respected. After all the nodes have been positioned, links are created between the nodes that are within each other's transmission range. The transmission range is a predefined value that signifies the maximum distance over which a message can be sent. The value for the transmission range is chosen arbitrarily and falls outside the scope of this research. For simplicity reasons, the transmission range was assumed to be global and symmetrical, meaning all communication in the network follows the same maximum transmission range. There is one

rule that is always enforced; the network must at no point become split into disjoint parts. If after the links were formed, it turns out this was the case, the network is discarded, and the nodes are scattered again.

With the network now created, we focus on the aspect of running traffic through the network. Because of the context, running the network manager on the server nodes was an obvious choice. The nature of the application makes the server nodes the focal point of communication. The extra computation power and energy supply of the server nodes make them able to run the computation needed by the GNN without any problems. Additionally, because of their role as data gatherers, they have an up-to-date network situation, which is needed for running the GNN. Making the server nodes the network manager also means having a centralized control scheme of the network. This means that any decision about how nodes are scheduled and routed is taken by the network manager. This makes sense because of the abundance of information the server nodes have. The way the network manager functions is mainly based on the type of application it is used for and the policy that is chosen. The policy decision is outside the scope of this work. However, it was needed for running the network. Since, in this case, the focus of communication is to bring information from the sensor nodes to the server nodes, this would create congestion closer to the server nodes. To address this, scheduling communication starts with the nodes next to the servers and then goes outwardly until the edges of the network. Additionally, communication is scheduled to avoid any collision issues. The routing is done simply through the shortest route possible from source to destination.

5.4 Training process

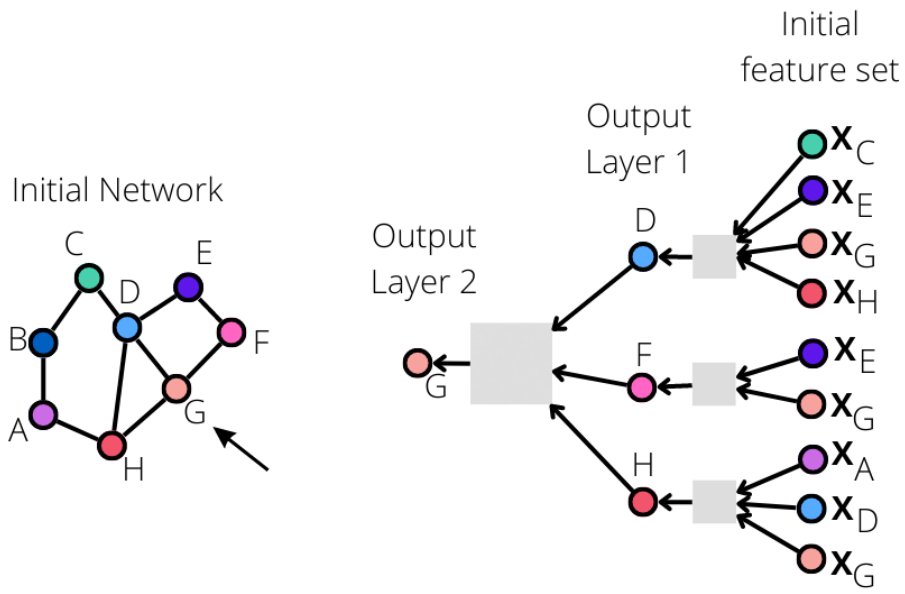

Figure 5.5: Process of creating the node embedding

The network environment needs to be given a network configuration. This configuration is the clustering of the network. With the configuration, the network manager assigns each node its respective cluster. Afterwards, the routing and scheduling for all nodes are decided. The information is disseminated throughout the network. The process happens before and separately from the normal functioning of the network. Time is simulated as a series of timeslots. Each

timeslot, the scheduled communication happens, and the counters of the nodes are updated. These counters are gathered at the end of the simulation period and used for statistical purposes.

The initial step of the process is extracting the information from the graph representation of the network. To achieve this, a preprocessing phase happens. In this phase, the network simulator is used. The network simulator initializes the networks of the given dataset. In the initialization process, each node is assigned a cluster through the Voronoi method, so depending on which server is closest. With this initial clustering, the network is run for a predetermined duration. Each node in the network has a predetermined pseudo-random period of time between messages. This period is determined when the node object is created and added to the network. The period is pseudo-random because its value is random within a certain interval. This interval is bounded for practical reasons. The lower bound is set such that the messages can actually reach the destination before a new one is sent; otherwise, congestion would be unavoidable. The upper bound needs to be at most the total duration of the simulation run. This measure is practical since there is no reason to schedule communication for time that does not exist. At the end of the first simulation duration, statistics are gathered to assess the Voronoi performance. These results are used in the evaluation process later on.

This initial clustering serves as a part of the initial feature set of the nodes. While conceptually, the input of the GNN is the entire network, in practice, this process is much more granular. The initial input layer is actually fed data node by node. For each data point of the node, an input neuron is defined. The input data is then processed like in Figure 5.5 to obtain the final node embedding. However, just the initial clustering is not expressive enough. The neural network would run into issues with training the model's weights. More precisely, too many nodes would have identical input data, leading to the optimizer constantly adjusting the same set of weights in hopes of getting better results. This is a problem because while the nodes may be in the same cluster, they are distinct and fit differently in the network topology. In the embedding process, the features of the neighbours of each node are used in calculating the final embedding of the node. As explained before, because the initial input would be the same, but the neighbours of the node would be different, the outputs would be different for identical inputs. This would confuse the optimizer and ruin the training process.

Choosing additional information to serve as input is a non-trivial task. This information needs to fulfil two main criteria. The first is that the data needs to be useful for the task at hand. The second is that the data needs to improve the identifiability of the node. The difficulty comes from the fact that the pool of available information from which to choose is small. Because distance plays such an important role in structuring the network, it became an immediate contender as the additional input data. However, obtaining precise coordinate data for each node is not only hard to get in real life but is also unnecessary. A much better approach is considering the hop distance in relation to points of interest. The points of interest, in this case, are the server nodes. The server nodes can be considered the pillars of the network. Since the server nodes are serving as edge servers for the network, they have fixed positions and have a wired connection. This makes them able to serve as reference points even in dynamic networks. The hop distance data is already available since it is used in the Voronoi process. The hop distance information does fit the two criteria. It is useful to the task because hop distance is a primary factor in choosing the clustering. Additionally, it improves the identifiability of each node. In essence, the hop distance of the node in relation to the server nodes is similar to a coordinate system. While this does not make nodes unique, it greatly improves their individuality from the perspective of the neural network.

Based on the input data fed initially to the GNN, it will output a soft clustering of each individual node. As described before, this soft mapping is simply the series of Softmax probabilities of each node to be part of each cluster. In order for the GNN output to be used by the network environment to run the simulation, the soft mapping Z needs to be turned into a hard mapping. The hard clustering then takes the form of:

$$hard(Z_n) = argmax(Z_n)$$

The cumulation of all $hard(Z_n)$ forms the new networks configuration $hard(Z)$. This new configuration $hard(Z)$ is then given to the network environment to reconfigure the network. Given

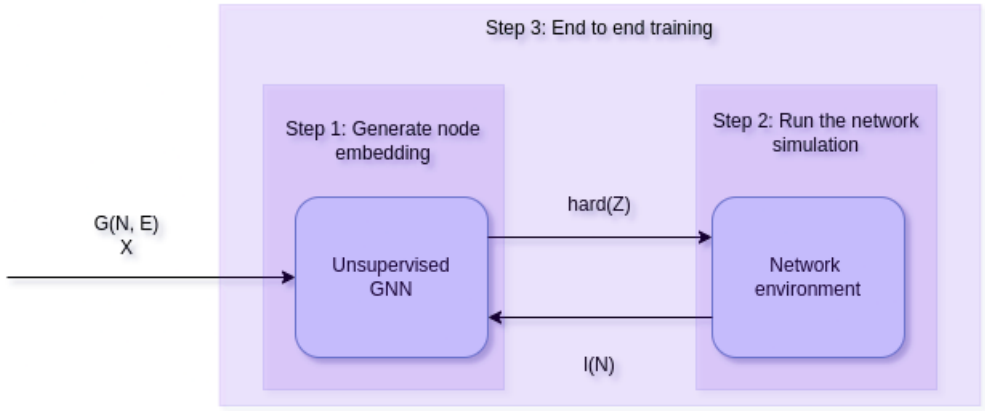

Figure 5.6: Training flow

this network configuration, the network manager schedules the appropriate communication according to their routing policy. Then the simulation runs for a predefined number of timeslots t. The actual size of a timeslot is set by the protocol used and is outside the scope of this research. At the end of the simulation, the network manager gathers statistical data about the network. The main focus is to create the vector $l(N)$ that holds the traffic load of each sensor node in the network. The network environment then returns the vector $l(N)$ to the GNN. With the given data, the GNN is able to calculate the loss and adjust the weights. For the role of the optimizer, Adam was chosen because of its wide use and generally good performance [50]. This whole training process is repeated on all the training mesh networks of the dataset for the predetermined number of epochs.

5.5 The program

The two algorithms presented in this section cover the use and execution of the implementation under normal circumstances.

Algorithm 1 presents how the training process of the GNN works. It is a step-by-step breakdown of the process. For more information on what each step does, please refer to Section 5.1 and 5.4.

Algorithm 2 encompasses the whole functioning routine. This algorithm includes the first one within it. The reason they are separated is to better visualize and understand the process.

Algorithm 1 The training process

Require: A dataset comprised of multiple mesh networks
Ensure: A trained GNN
for every network in the dataset **do**
    Initialize the network environments
    Initialize the parameters of the GNN
    Obtain GNN input $G(N, E)$ and feature matrix X from the network environments
end for
for epochs **do**
    for every network in the training dataset **do**
        Generate embedding Z from the GNN
        Transform Z into hard(Z)
        The network environment configures the mesh network according to hard(Z)
        Run the simulation environment for t timeslots
        Obtain the network load vector l(N)
        Compute the loss according to the loss function
        Train the weights through backpropagation
    end for
end for

Algorithm 2 Functioning under normal circumstances

Perform GNN training
while Wireless mesh network is functional **do**
    Let the mesh network run for t timeslots
    Obtain the snapshot of the current network
    Feed the snapshot information as input of the GNN
    Generate embedding Z from the GNN
    Transform Z into hard(Z)
    The network environment configures the mesh network according to hard(Z)
end while

5.6 Discussion on expectations and limitations

The main limitation of our model comes from the context of the problem and the nature of GNNs. Clustering can be translated as a node classification task for the purposes of GNNs. The problem appears when looking at the graph structure overall. Through simple node classification, there is no guarantee for cluster integrity. The case will appear where nodes of a cluster will appear within another cluster. The cause for this is the influence of the neighbourhood of the node. In the case of networks, this is highly undesirable. While networks are usually represented as graph structures, they are not the same. Positions of nodes matter for network representations. However, GNNs on their own do not account for this.

To address the problem, we added an element to the loss that takes the position of nodes into account. The added element needs to outweigh the other component of the loss to maintain cluster integrity. However, this approach is rigid and limiting. Because of this design decision, the performance of our model is expected to be comparable to Voronoi. While our model is still expected to beat Voronoi overall, the wide range of use cases for the model made us implement a more generalized solution. By restricting the problem space, a very specialized model could be created for certain types of topologies. Conversely, one could take the approach and change our model to be more adaptive to the networks it is used on. One such approach could be implementing Reinforcement learning for training the weights of the GNN model. The networking space can be easily modelled to fit the requirements for Reinforcement learning.

One thing that our solution will leverage is the neural network aspect. The expectation is that performance will be better on datasets with a high degree of similarity between networks. The neural network would have an easier time learning the patterns of such datasets and would perform better.

Because of the spatial nature of the GNN layer used, our model should perform better on networks with a high level of connectivity or networks with high density. The level of connectivity here means the average degree of nodes in the network. Density is the number of nodes per unit of area. In this case, high density would be a high number of nodes over a small area, which can happen, for example, in the case of points that require multiple sensor nodes to collect data. The increased number of connections translates to a more radicalized clustering. Nodes with a lot of neighbours in the same cluster will become even more strongly part of that cluster, while nodes on the extremity of clusters will become easier to influence to change clusters. A drawback would be the added computational cost of the increased connectivity.

The computational complexity of our model, and GNN models in general, might be the biggest limitation. Scaling up for GNNs is not linear from the perspective of computational complexity. In the case of our model, for example, the change from 2 server nodes to 4 with all other things being the same would result in more than four times as many needed computations. Doubling the number of nodes in the network will result in at least four times the computation duration, mainly due to the added complexity of the bigger adjacency matrix. The problem with computational complexity is aggravated by the general drawbacks that normal neural networks suffer from, namely, the cost (both monetary and time-wise) of assembling an expansive and viable real-world dataset to use for the neural network; the lengthy amount of time needed to train the neural network.

Chapter 6

Experimental results

6.1 Experimental setup

6.1.1 Software and Hardware

All the tests in this section were run on an Acer Spin 3 laptop with a quad core Intel i5 8250U processor and 8GB of RAM. The version of Python was 3.8.10. The Pytorch version was 1.10, the CPU edition. The Pytorch Geometric version used was 2.0.2.

6.1.2 Research scenarios

The scenario is based on the intelligent agriculture applications that acquire environmental data through multiple sensors like temperature detectors, infrared cameras, humidity sensors, etcetera. This scenario encompasses a wide variety of cases and topologies. While the requirements for this scenario are relatively forgiving, the multitude of cases to cover can prove challenging.

In this scenario, the network is composed of numerous IoT devices and several servers. The network is divided into clusters, with each cluster containing exactly one server. The normal IoT nodes in the cluster send data to the server. These IoT nodes monitor environmental parameters such as temperature, humidity, movement, etcetera. Wireless communication is centrally managed by the network manager. All nodes have to periodically send data to the servers. However, this period does not have to be identical for all nodes. Some nodes need to report more frequently than others.

The topologies can vary in this scenario. While some networks can be very structured, like a field of grain, some can be more chaotic and random, like crops in a greenhouse. Some networks have different sensor distributions, with high and low density areas, like an orchard. Additionally, mobile elements can be present in the network, like farming equipment. This led to the need for a multitude of testing topologies. The size of these networks also can vary wildly and should be taken into account

6.2 The datasets

Multiple datasets have been used in testing the implementation. Each dataset is comprised of a set of IoT networks split into two subsets, one for training purposes and one for testing purposes. All of the topologies for the IoT networks are generated in accordance with the rules of each dataset.

The first dataset is also the most varied, namely the fully random dataset. This dataset is comprised of 100 training networks and nine testing networks. There are no set rules for this dataset. The networks of this dataset vary in both size and topology style. However, the size of the networks is limited to medium sized networks, between 20 and 70 nodes. Because of this high

amount of variability, the results on this dataset are expected to also vary wildly between runs of the program.

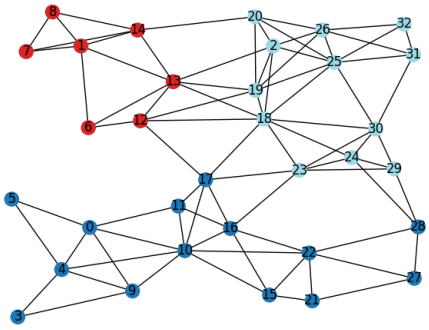

Figure 6.1: Random network example 1

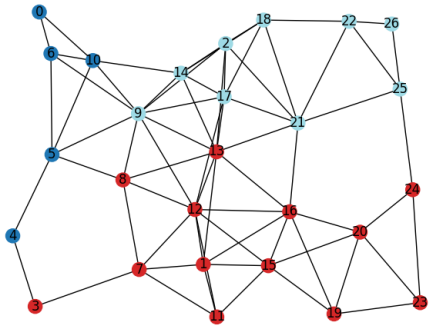

Figure 6.2: Random network example 2

The second dataset is the dynamic dataset. The idea behind this dataset is to observe the behaviour of IoT networks in time. This means that the first network in the dataset is the original network, and each of the subsequent networks in the dataset is a time snapshot after the original network. Each snapshot has one difference compared to the previous in the dataset. One random node in the network changes its position to a new random location, thus creating the snapshot. The new location of the node needs to ensure that the network does not suddenly become disjointed. Running on this dataset would create a very specialized neural network for this specific topology. This dataset is comprised of 20 training networks and ten testing networks. The low number of training networks is because of the low variability between networks of this dataset.

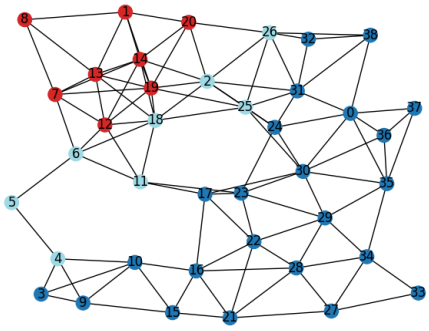

Figure 6.3: Dynamic network example 1

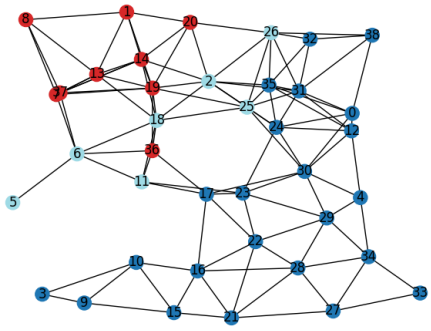

Figure 6.4: Dynamic network example 2

The third dataset is the grid dataset. All the networks in this dataset follow a strict grid structure with fixed positions and distances for nodes. The server nodes also have fixed positions along the main diagonal of the grid rectangle. The training dataset has 50 networks, and the testing one has 14 networks.

The fourth dataset is the net dataset. This dataset is similar to the grid dataset but with more loose restrictions. The nodes are scattered in a loose grid fashion, meaning that each node has some variability as to its exact position. However, they can generally be expected to be in a certain area. This change leads to the creation of several links between nodes that adds to the variability of the networks. The server nodes follow a similar trend to the normal nodes. They

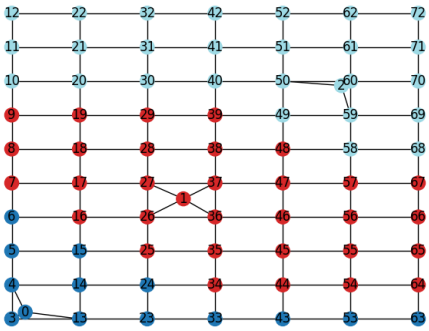

Figure 6.5: Grid network example 1

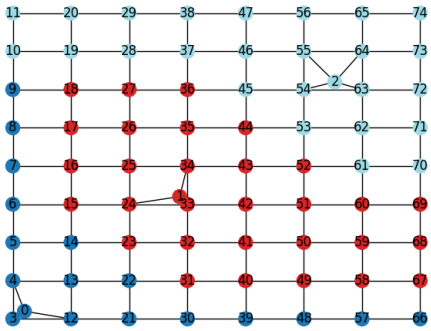

Figure 6.6: Grid network example 2

each have a certain area where they could be found, but the precise position within this area is randomized. The training set is comprised of 50 networks and the testing set of 10 networks.

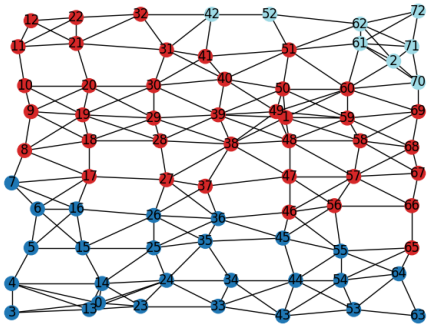

Figure 6.7: Net network example 1

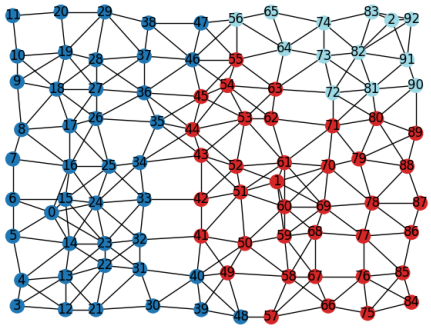

Figure 6.8: Net network example 2

The fifth dataset is the sparse, dense dataset. This dataset follows from the net dataset. The majority of nodes are spread out like the net dataset. However, a certain percentage of nodes is distributed on top of the previously set nodes. This redistribution creates areas of high node density that emulate real-world networks with places of interest that require more sensors. The dynamics of this dataset are completely changed since the high density areas create spots with a high flux of data and possible congestion. The training set is comprised of 50 networks and the testing set of ten networks.

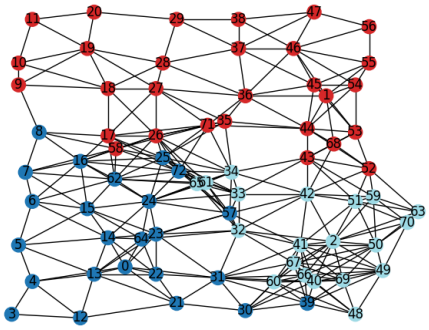

Figure 6.9: Sparse-dense network example 1

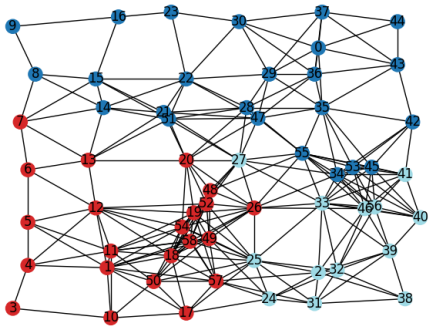

Figure 6.10: Sparse-dense network example 2

6.3 The evaluation metric

For the evaluation process, we have defined an overall load balancing metric to make comparison easier. This metric summarizes the load balancing metric of each cluster. Given the formula defined in Chapter 2 for the load balance of each individual cluster, the overall formula for load balancing would be the Frobenius normalization of the vector of all individual load balancing values.

$$B(c) = |L(c) - \bar{L}|$$

$$J = ||B(C)||_F$$

The reason for choosing the Frobenius normalization for this metric has to do with the curve of the function. The average of the results is too linear in progression. The average of the squares has the opposite issue of growing too fast. Frobenius normalization has the benefit of being easily influenced by larger values. This translates in actually lower overall load balancing values for load values that are closer to the mean load of the clusters. For example, with 3 clusters and two cases, B(1)=2, B(2)=7 and B(3)=9; B(1)=4, B(2)=5, and B(3)=9; the second case is slightly better balanced than the first one and will result in a lower overall load balancing score.

As is to be expected, the value of this metric is highly dependent on the number of nodes in the networks and the number of server nodes. The overall load balancing metric will be 0 or close to 0 for the optimal clustering solution of the network. The higher the metric results then, the worse the clustering performs. However, there is no practical "worst case" result; the function is unbounded on one side. With these in mind, for orientation purposes, it can be considered that for small networks (<20 node), if the overall score is below 5, the clustering is well balanced; for medium sized networks (>20 nodes and <70 nodes) if the overall score is below 10, the clustering is well balanced; for larger networks (>70 nodes) if the overall score is below 15, the clustering is well balanced.

6.4 The tuning of hyperparameters

The process of optimizing hyperparameters is very important in tuning the performance of any neural network. The first step is to identify what hyperparameters actually impact the output of the model. Six hyperparameters were tested: learning rate, number of epochs, number of neurons in the hidden layer, the value of the weight in the objective function, dropout rate and activation function. All the tests were run on the random dataset to avoid any possible advantage and get the a multitude of results. For each test, the model was run three times with different random seeds, resulting in different network topologies in the dataset. The results were then averaged

to obtain the final result for that particular test. The dropout rate was swiftly eliminated from testing since it showed no differences between runs.

The first step is to establish a baseline. This default set of hyperparameters will be used as the starting point for the tuning and also for comparison purposes. After the baseline is set, each of the individual parameters that have to be checked will be tested in isolation. This means that only one particular variable will be changed at a time in each test. This way, we can isolate the effects of just that particular change. For each of the tested attributes, a range of values needs to be established that would offer a wide enough spectrum to account for multiple possibilities but also granular enough to not miss important chances by skipping them. After the initial isolated elements are tested, the best value obtained for each attribute will be selected. With these selected cases, combine hyperparameter tuning will commence. In this process, the combinations of the selected cases will be tested.

As a preparation, a few preliminary tests were run just to establish the ranges of the hyperparameters. The results of these indicated that the baseline hyperameter set should be set as follows, the activation function should be ReLU, mainly for its widespread use and simplicity; the learning rate should be 0.001, the number of epochs should be 50; the number of neurons in the hidden layer should be twice the number of input features, and the weight of the loss function should be 0.03. These values provide a good starting point for this analysis.

6.4.1 Activation function comparison

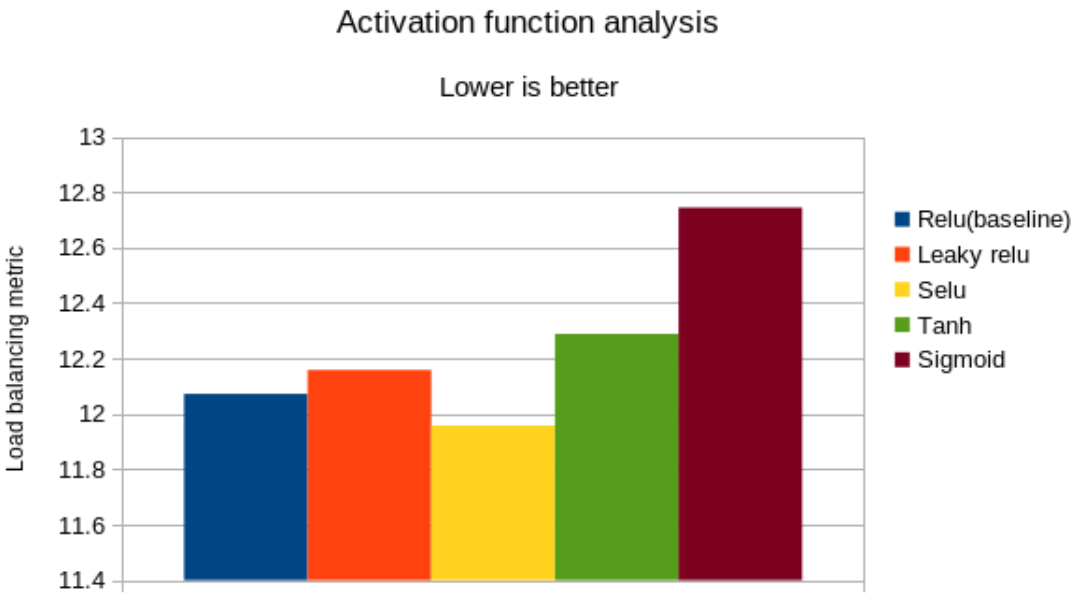

Figure 6.11: Activation function comparison

The activation function tested were ReLU, Leaky ReLU, SeLU, Sigmoid and Tanh. While there were slight performance differences for these tests, they were marginal, as can be seen in Figure 6.11. Given this fact, testing the activation function was eliminated from the list of tests for further purposes. This means that the final choice for the activation function remains ReLU.

6.4.2 Learning rate comparison

Learning rate is an important factor that dictates the step size the neural network will make in trying to learn and understand the data. In general, the expectation is that the value of the

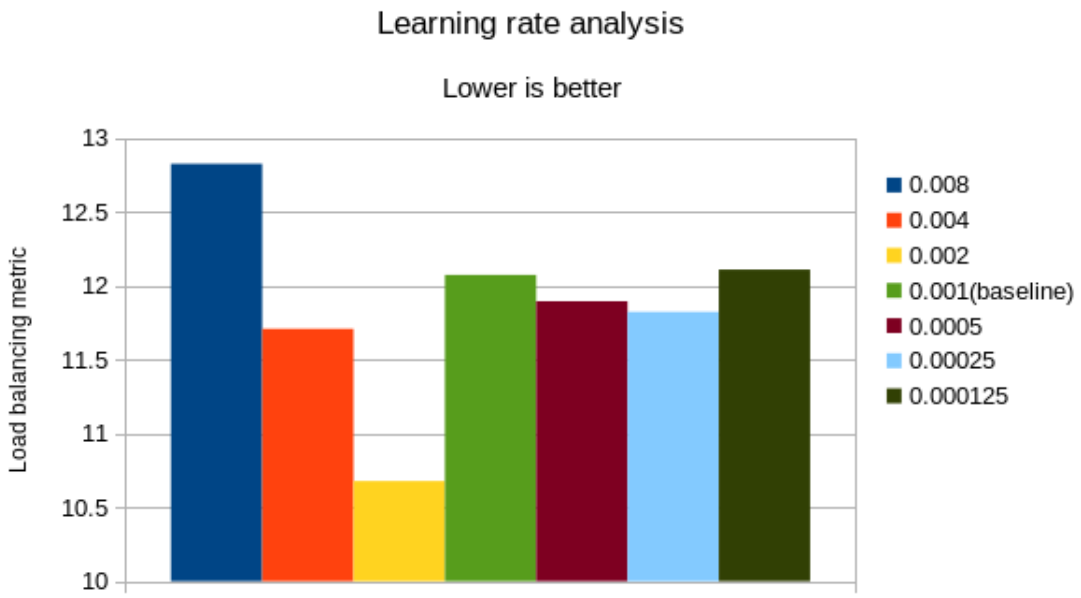

Figure 6.12: Learning rate comparison

objective function will eventually converge, meaning it will go down sharply and reach a state of semi-stability, where additional learning will only provide marginal improvement. However, if the learning rate is too high, in its process to learn, the neural network might actually overshoot, going over the convergence stage. There is a trade-off to be had here between a low learning rate, which will require more time and computational power to reach good results, and a high learning rate, which will reach the goal faster, but might overshoot it in the process. For this test, we have chosen the set of values starting from the baseline value by dividing or multiplying by 2. This was done until almost a whole new degree of magnitude was reached. However, due to the nature of the learning rate, this process could not be done fully in isolation. Since the learning rate is so intrinsically linked to the number of epochs given to the training process, it would have been unfair for some values to maintain the same number of epochs as for the other tests. As such, the number of epochs was adjusted to give all tests a fair chance.

From the results, shown in Figure 6.12, we can notice that learning slower does not provide any significant benefit. If anything, it is detrimental due to the increased number of epochs that would slow down the whole learning process. It can be seen that learning just slightly faster does provide the best results. This could be attributed to the low number of both neurons and nodes in the networks. However, it likely is the case that going faster than this simply overshoots the optimal results seeing that, for the case of the highest learning rate, the results are by far the worst of the tested values.

6.4.3 Number of epochs comparison

The number of epochs represents the length of the training process or how many times the neural network is allowed to go over the network in hopes of learning the task that it is supposed to do. With the results, Figure 6.13, we can observe that given too much time, the neural network will simply start overshooting, the learning process starting to resemble a parabola. As a matter of fact, the diagram for this test resembles a normal learning process diagram. We can see the sharp performance increase as the epochs increase up to a point and then the decrease in performance as the number of epochs increases further. The best case is given when the number of epochs is slightly increased compared to the base case, namely 100 epochs. This result basically confirms

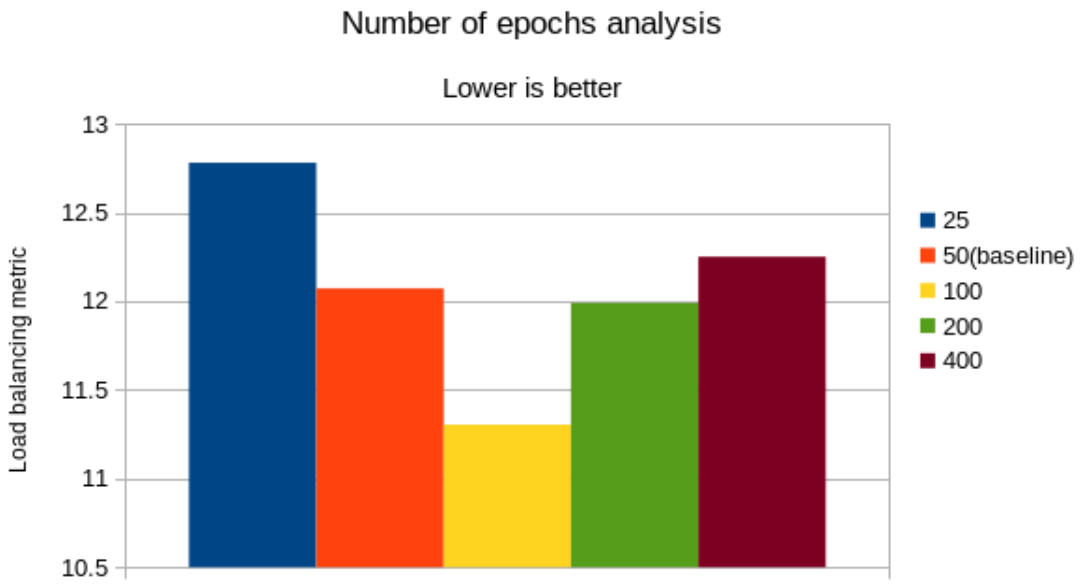

Figure 6.13: Number of epochs comparison

the results for the learning rate. The graph neural network needs either more time to learn or needs to learn faster. The results show that it is actually better for the model to learn faster than it is to give it more time.

6.4.4 Number of neurons comparison

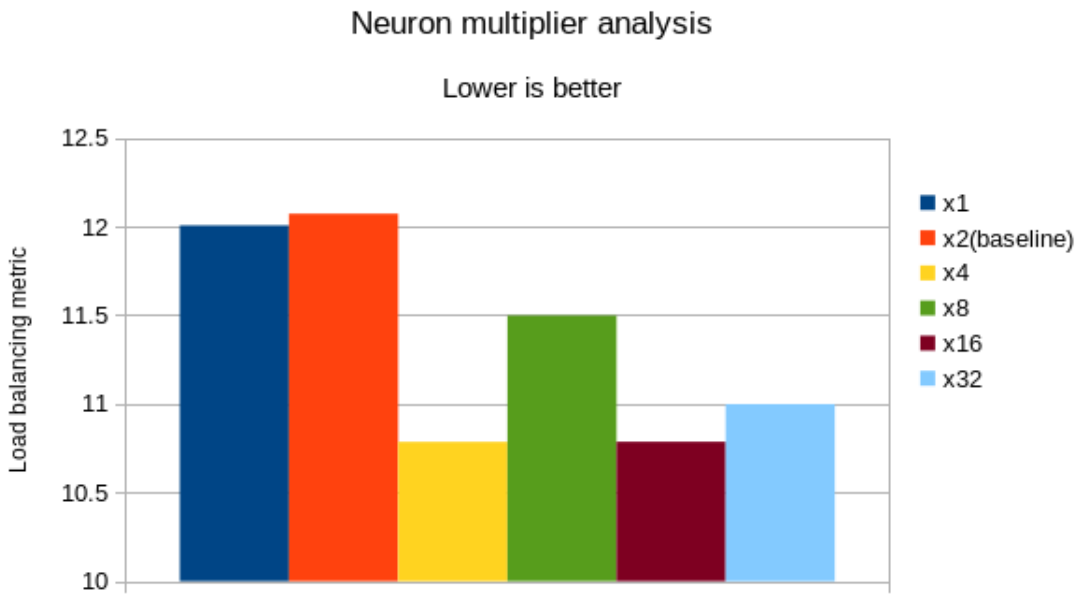

Figure 6.14: Neuron multiplier comparison

The number of neurons directly dictates the structure of the neural network. While the number

of input and output neurons cannot be changed because they are dependent on the dataset and the task, the number of neurons in the hidden layers can be altered to suit the performance better. Due to the design choice of having only two GNN layers in the model, there is only one hidden layer that can be tuned. This simplifies the tuning process. Given the nature of the task at hand, namely that shape of both the input and output layers are intrinsically linked to the number of server nodes in the IoT network, and how this number of servers can vary based on the dataset, the choice was made that instead of having a flat, static number of neurons in the hidden layer, the number of neurons should also be dependent on the server number. More precisely, the number of hidden neurons is linked to the number of input neurons. This means that the number of hidden neurons is always going to be a multiple of the input neurons. In both diagrams and for the rest of this paper, the number of neurons will be represented as a multiplier, for example, times 2. Another choice was to multiply the number of neurons and not divide. This is because the number of input neurons is already low. Dividing it further would not be optimal. Additionally, the division would add the problem of numbers that are not perfectly divisible, which would require another design choice of how to deal with them.

When it comes to the results, Figure 6.14, it is clear that the model would benefit from an increased number of neurons. This can probably be attributed to the fact that it would allow the model to categorize nodes better, in a more nuanced way, since now it would have more choice. Crowning the winner in this category is a bit harder; both x4 and x16 have practically identical results. In the end, the choice was made to put forward x4 as the best for the number of neurons. The decision came down to the fact of added computational complexity. Since the number of calculations needed to be made in the training process is directly dependent on the number of neurons, x16 was simply more computation for no apparent benefit.

6.4.5 Loss weight comparison

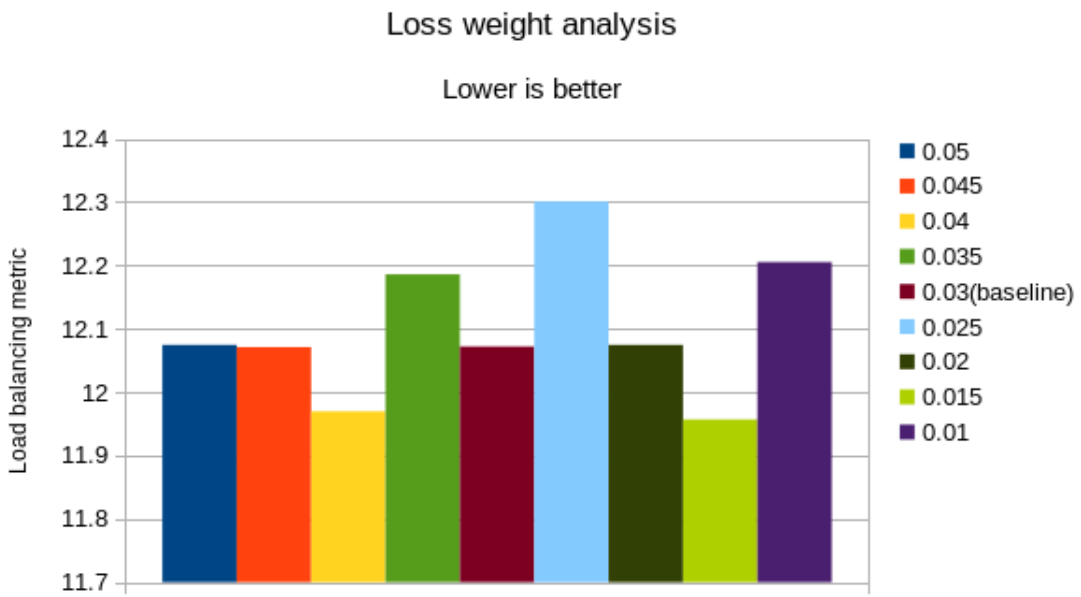

Figure 6.15: Loss weight comparison

The weight of the loss function dictates the proportion in which each individual loss impacts the overall results. As explained in the previous chapter, this weight ensures that the model functions correctly. This was a critical factor in deciding the spectrum of values. For values above the ones tested, the load balancing part of the loss function can become bigger than the hop distance part.

This actually will not be noticed in the load balancing results themselves but will be noticeable because of nodes that will be visibly mismatched to their cluster. Conversely, if going lower than these values, the impact of the load balancing part becomes so small that the implementation basically becomes the Voronoi solution. From a results perspective, Figure 6.15, it can be noticed that the impact this variable has on the performance is rather minimal. While not as small of an impact as the activation function comparison, even the difference between the best and worst of the tested cases was less than half a point on the load balancing scale. This can be attributed to the limitation in values and them being bounded to such a narrow spectrum. This is unfortunate, but it speaks to the limitations of the approach. In order to maintain a valid output while using the current method, this is a trade-off to be made. The best result was when the weight was 0.015, which will be used in the next section of the tuning.

6.4.6 Combined hyperparameters

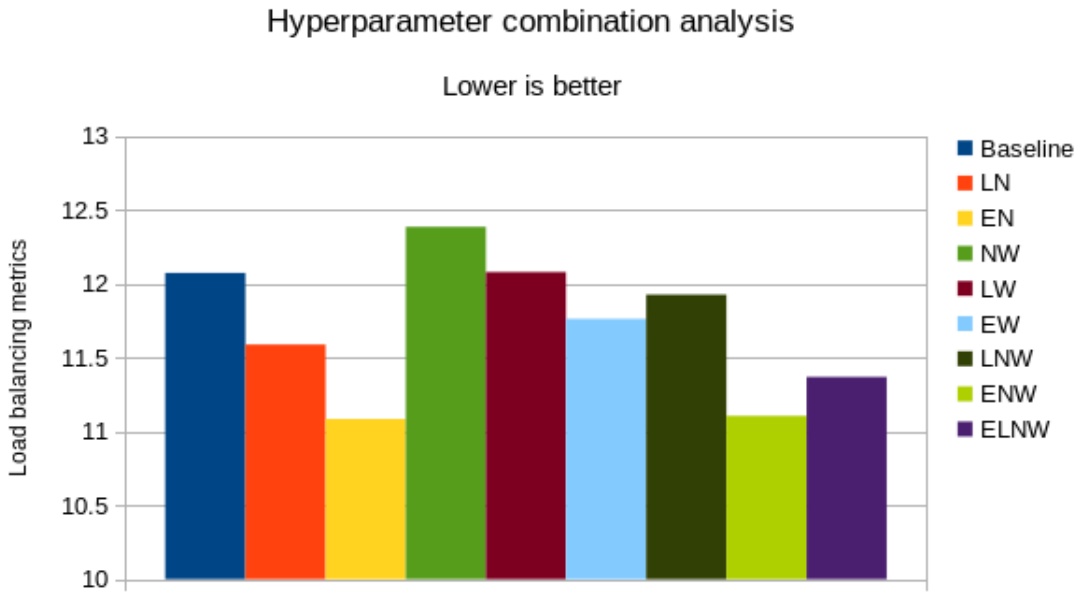

Figure 6.16: Combined hyperparameter comparison

Given the best choices for the individual hyperparameters from the previous tests, we now start combining them to observe their interactions and results together. An important note to make is that, since learning rate and the number of epochs are intrinsically linked, and they serve a very similar purpose overall, most of the tests where both of these would be present were skipped. The assumption here was that they would make the model both learn faster and for longer, thus making it more likely to overshoot and hindering performance. In Figure 6.16 and for the rest of this paper, the choice was made to identify each combination of hyperparameters by code. Each code is an abbreviation of the variables that are changed for that particular case. L stands for learning rate. E stands for the number of epochs. N stands for the number of neurons. W stands for loss weight. As explained before, activation function and dropout rate were eliminated because they provided little to no change at all in results. From the results, we can observe that all the combinations that include the number of epochs and number of neurons simultaneously performed well. This does somewhat makes sense since one would be paring the potential for more nuanced relations between input and output due to the number of neurons with the longer time to learn these relations.

6.4.7 The best setting

Taking into account all the previously presented tests, multiple settings showed potential. There are a few to rise above the rest. Of note here, there are 2 with practically identical performance. These are the best case for the learning rate and the best case for the number of neurons. While the combination between the number of epochs and the number of neurons came close, it was the third in terms of performance after these two. The decision must be made then, which is going to be used for the rest of the experiments. In the end, the deciding factor was again computation time. Increasing the number of neurons also increases the computational complexity. The best setting can be seen in Table 6.1

Hyperparameter	**Value**
Activation function	ReLU
Learning rate	0.002
Number of epochs	50
Neuron multiplier	x2
Loss weight	0.03

Table 6.1: The best setting for hyperparameters

6.5 Results on random dataset

The random dataset provides a very important challenge. Because of the wide variety in both topologies and sizes of the IoT networks, relations between nodes and clusters might be harder to establish. This is a clear disadvantage for any machine learning method. A total of ten tests were run on this dataset. For each run, the composition of the networks changed and/or the initial distribution of the weight for the model. While they were allowed to vary in size, the IoT networks for these tests were limited to medium-sized networks with three server nodes. The result of each run represents the average of the load balancing metric of all the networks in the testing set, seen in Figure 6.17. These results are then further summarized in the average performance of each individual implementation of this dataset, seen in Figure 6.18.

Given the difficulty of this dataset, the fact that our model managed to have the overall best performance was a pleasant result. Even looking at individual runs, we can notice that, at worst, our model performs as well as Voronoi. As can be seen from the performance of the Tsitsulin implementation, their performance is worse than Voronoi's, which was more in line with the expectation of this dataset. The results can be then attributed to two aspects. The first reason has to do with the nature of the dataset itself. Because of the nature of randomness, there will be networks in the dataset that favour one of the implementations. However, when averaging the results, all of these end up balancing each other out. This also would explain the rather small overall differences in the end results between the different implementations. The second reason has to do with how our implementation works. Because of the loss function, the model is set up to achieve its goal by basically exchanging the fringe nodes that are far enough away from servers, which is done to ensure cluster cohesion. However, it also means that performance-wise there will be a high similarity to Voronoi, which assigns cluster membership solely on proximity to the servers. The main difference comes with the optimization of trying to achieve better load balancing, making our implementation generally perform better than Voronoi.

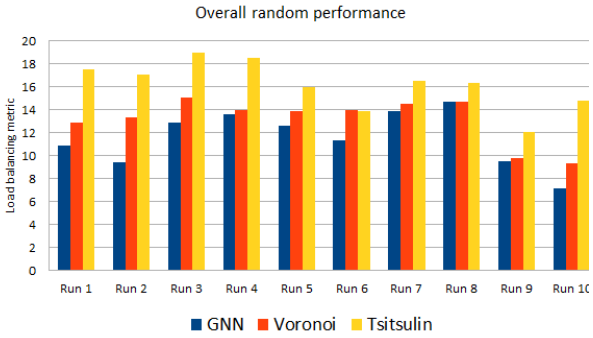

Figure 6.17: Overall random performance

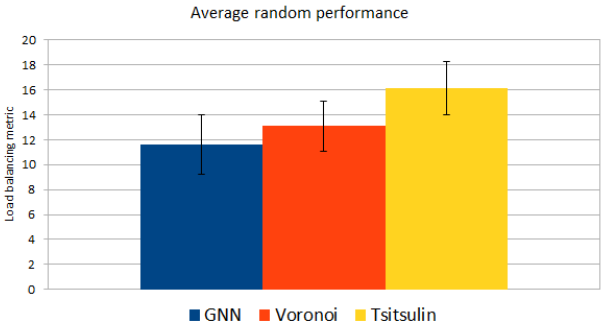

Figure 6.18: Average random performance

6.6 Results on dynamic dataset

Time is a very complicated and complex thing to model. There are multiple ways to be represented depending on the context in which it is used. According to the taxonomy defined in [51], the type of networks we are dealing with in our setting are Discrete node-dynamic evolving networks. Discrete mainly deals with how time is represented, in this case, in a discrete manner. This means that the behaviour of the network can be captured in periodic snapshots of the network. In the case of our setting, this is appropriate since nodes do not move that often, making the network snapshots able to accurately capture the topology of the network. Node-dynamic means nodes can move in between snapshots. Evolving means links of the network, while changing, follow a slower change process. In our context, the links themselves do not change on their own but rather change because of the movement of the nodes. There is one important distinction to be made because of what our links represent, which is simply the ability of two nodes to communicate with each other. The snapshots of the network can be treated as standalone networks. The relation the network links represent is too trivial, and no information can be gained through a temporal element added to the GNN.

In this dataset, GNN methods should have an inherent advantage because of their ability to learn and given the high level of similarity between the networks in this dataset. A total of ten tests were run on this dataset. For each run, the composition of the networks changed and/or the initial distribution of the weight for the model. The result of each run represents the average of the load balancing metric of all the networks in the testing set, seen in Figure 6.19. These results are then further summarized in the average performance of each individual implementation of this dataset, seen in Figure 6.20.

The results confirm the initial expectation. Both GNN based methods heavily outperform the naive solution. However, it is very important to point out that this dataset is very dependent on the initial network. As can be seen from the different runs, if the initial network is favourable to one of the implementations, most, if not all, of the rest of the networks in the dataset will be favourable. While this makes sense, it results in this test having the highest standard deviation for the average results from all the tests.

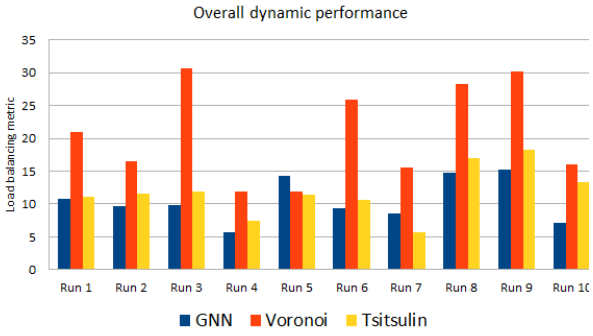

Figure 6.19: Overall dynamic performance

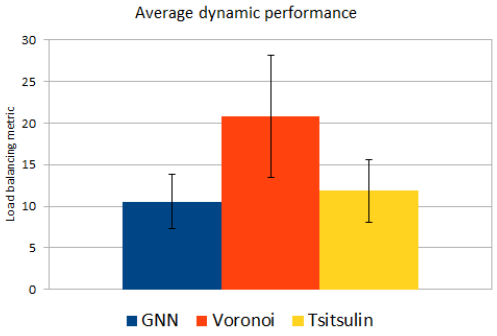

Figure 6.20: Average dynamic performance

6.7 Results on grid dataset

The grid dataset provided a completely different set of requirements. While the size of the networks continued to vary, just like with the random dataset, the type of topology was similar. Because of this consistency in characteristics, neural network approaches should be able to better infer relations in the data. A total of ten tests were run on this dataset. For each run, the composition of the networks changed and/or the initial distribution of the weight for the model. The result of each run represents the average of the load balancing metric of all the networks in the testing set, seen in Figure 6.21. These results are then further summarized in the average performance of each individual implementation of this dataset, seen in Figure 6.22.

From the results, we can see that the previously mentioned expectation did happen. Both GNN based implementations performed best, their individual performance being practically identical. However, it can be said that the defining reason why the GNN implementations did well on this dataset is not necessarily the similarity of the networks in the dataset but the highly structured nature of the networks. Additionally, because of the shape of the networks, they cannot be easily balanced based on just the distance to the servers. This helps to explain particularly why Voronoi performed poorly.

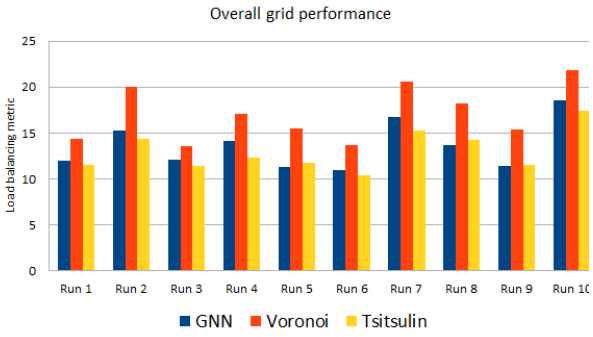

Figure 6.21: Overall grid performance

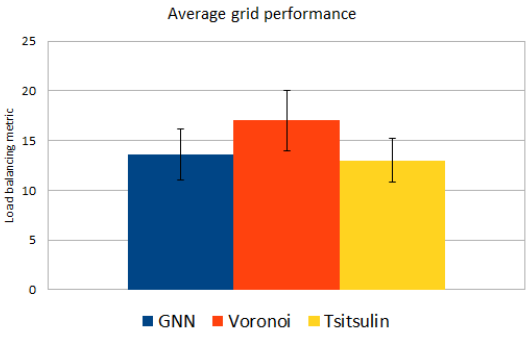

Figure 6.22: Average grid performance

6.8 Results on net dataset

The net dataset was a high similarity to the grid dataset. This does make a lot of sense, considering the net dataset is derived from the grid dataset. While it has an element of randomness in it, the overall structure of the dataset is stable. Nodes will still be in roughly the same places with

a very similar connection pattern. While some links between nodes might be created or deleted, the overall topology remains similar. The size of the networks can vary, just like with the grid dataset. However, the additional change that the position of the server nodes has a higher level of freedom will impact the results to a certain degree. The reason behind this is that it can shift the natural load distribution of the network. Because of the similarity to the grid dataset, neural network approaches should be able to better infer relations in the data. A total of ten tests were run on this dataset. For each run, the composition of the networks changed and/or the initial distribution of the weight for the model. The result of each run represents the average of the load balancing metric of all the networks in the testing set, seen in Figure 6.23. These results are then further summarized in the average performance of each individual implementation of this dataset, seen in Figure 6.26.

Confirming our expectation, we can observe that the shape of the results is similar to the results of the grid dataset. This also confirms the idea stated previously that a high level of structure is preferred by the GNN approaches. This is especially true for the Tsitsulin approach, which, while just slightly, has managed to perform better than our approach on this dataset.

The most important aspect that was learned from these tests is the type of network our implementation preferred. While it is true that the common aspect between both the grid and net dataset when compared to the random one is the structured nature, this can be misleading. There are other similarities that can be overlooked if not paying close enough attention. Other similarities include the fact that, naturally, these datasets have well-connected nodes and the fact that their natural topology creates a load unbalance. These aspects are going to be explored in the next section.

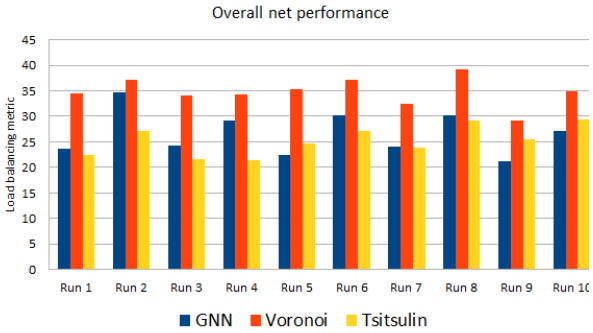

Figure 6.23: Overall net performance

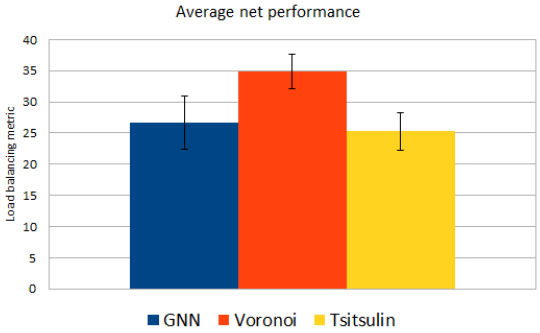

Figure 6.24: Average net performance

6.9 Results on sparse-dense dataset

The sparse-dense dataset presents networks with a high level of diversity. While the underlying structure is that of the net dataset, the redistributed nodes that create dense locations totally change the dynamics of the dataset. The density of an area is defined by the amount of nodes present in it. A sparse density area is defined by the almost uniform distribution of nodes over it, with ample space between nodes. A dense area is defined by the very high amount of nodes present the are very close to each other, thus breaking the minimum distance rule mentioned is Section 5.3. The sparse-dense distribution is then defined by the percentage of nodes that are redistributed to form the dense areas. Considering that the redistributed nodes also follow a mostly random pattern, this dataset can also be seen as a combination of structured and random elements. For these initial tests, the redistribution proportion was set to around 20% of the size of the network. This means that around 20% of the nodes were scattered over the net structure to create dense pockets. Because of the increased interest, a total of 14 tests were run on this dataset. For each

run, the composition of the networks changed and/or the initial distribution of the weight for the model. The result of each run represents the average of the load balancing metric of all the networks in the testing set, seen in Figure 6.25. These results are then further summarized in the average performance of each individual implementation of this dataset, seen in Figure 6.24.

The results of these tests have proven quite interesting. Not only did our implementation win overall, but compared to the Tsitsulin implementation, it performed more than twice as good. This gave credence to the observations made during the net dataset testing. Our implementation seems to prefer well-connected unbalanced networks. These aspects due make sense conceptually. The unbalanced part is simple to explain; if the networks are naturally well balanced through just Voronoi, there is not much that can be done to improve that, even though using neural networks. The second aspect has more to do with how GNNs work. Conceptually, for GNNs to create the final embedding of a particular node, they require the information of its neighbours. This makes well-connected nodes able to pull information from more sources to help create their final embedding. This can also be detrimental in some cases. If a node is influenced by too many sources, the meaning of its embedding is diluted, which will impact overall performance results. This is the reason why the design decision was taken to limit the number of GNN layers to just two. More GNN layers would just pull information from nodes too far away that are less relevant or can be even harmful to the performance. In practical terms, the nodes that we want to change clustering to obtain better load balance are the nodes at the extremities of the clusters. Having more connections in these areas results in a more diverse set of influences. This makes it easier for nodes to switch clustering in order to achieve the model's objective.

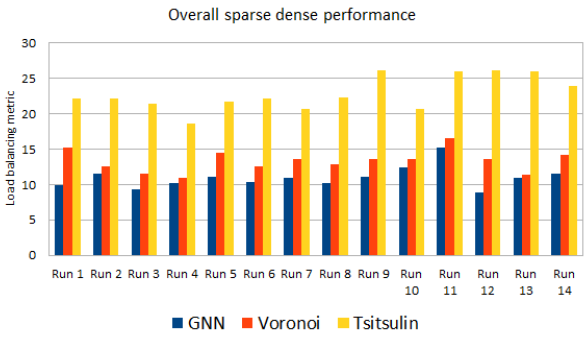

Figure 6.25: Overall sparse-dense performance

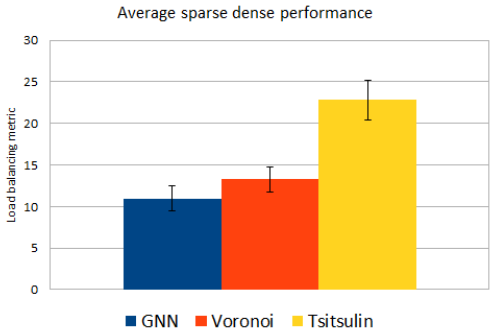

Figure 6.26: Average sparse-dense performance

6.10 Comparison results of different sparse-dense distributions

Considering the findings in the last tests, there is now an interest in finding how this phenomenon progresses under different conditions. The main factor is the proportion of redistributed nodes. This is basically what makes the sparse-dense dataset special. The progression of this percentage is studied in this section. The decision was to start with no node redistributed and progressively get to the 50% point. It is important to point out that the way server nodes are placed different compared to how the net dataset does, which is the reason why the results for 0% are different from the net ones. It was considered that there is no point going over 50% redistributed nodes since, at that point, the network simply becomes a smaller, denser and more compact network. There are no more sparse areas to create contrast. Examples of how the networks would look at under different distributions can be seen in Figures 6.27 - 6.31

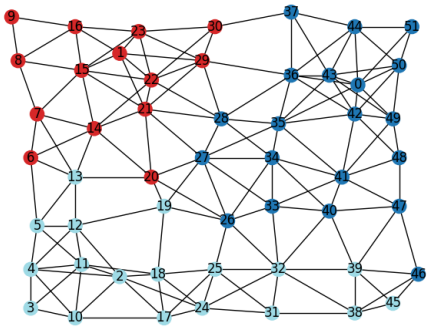

Figure 6.27: 0% Dense network example

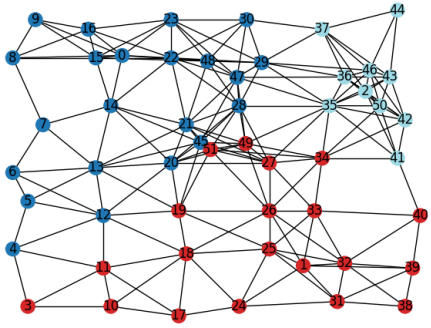

Figure 6.28: 15% Dense network example

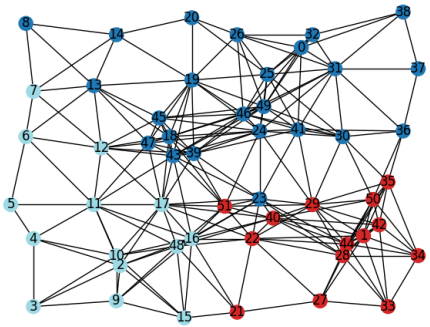

Figure 6.29: 27% Dense network example

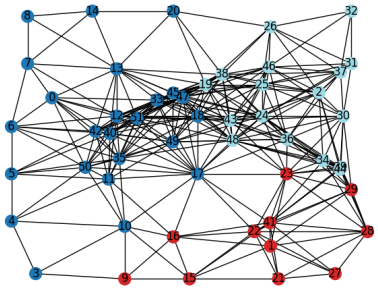

Figure 6.30: 40% Dense network example

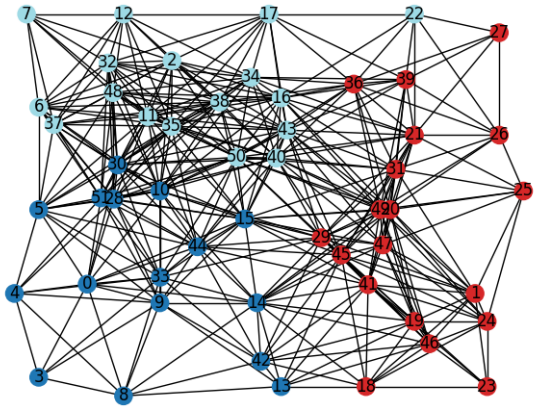

Figure 6.31: 50% Dense network example

In Figure 6.32, the overall results of these tests can be seen. Each of the tests represented in the figure is the average of five individual runs under the given conditions. For each run, the composition of the networks changed and/or the initial distribution of the weight for the model. It can be observed that the results for our implementation actually remain mostly the

same throughout the tests. However, the results of all the other implementations vary throughout the tests. In all cases, our implementation comes out on top. These results do show that our implementation performs well despite the changes in the topology. This does confirm our previous expectations. The performance of our implementation, in this case, is more evident through comparison with the different implementations.

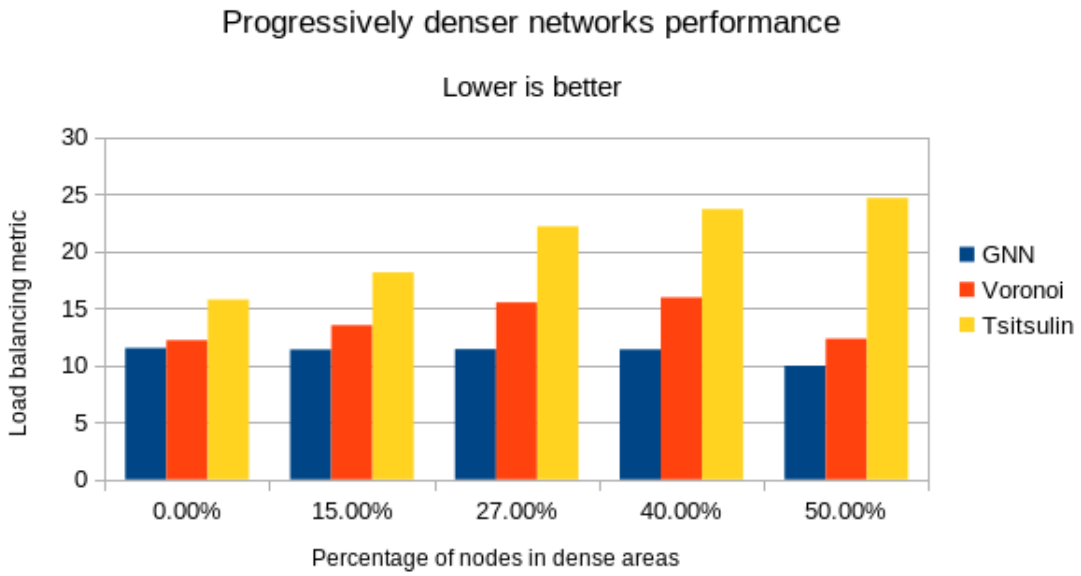

Figure 6.32: Progressively denser networks performance

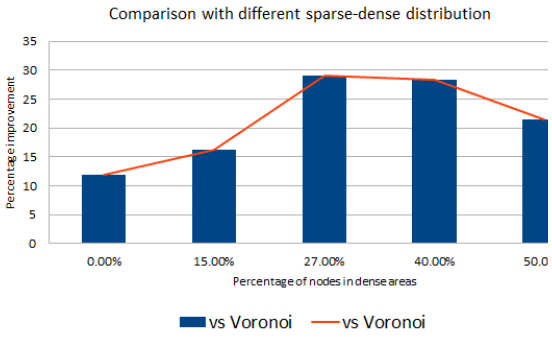

Figure 6.33: Trend line comparison with Voronoi

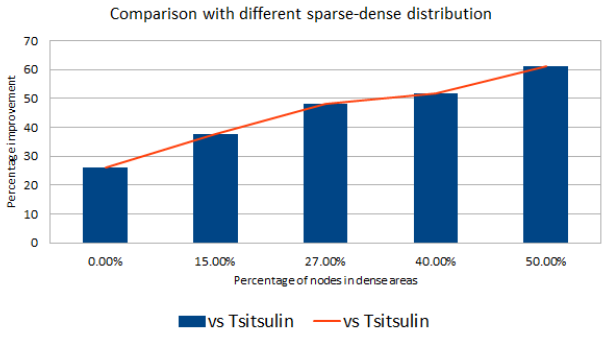

Figure 6.34: Trend line comparison with Tsitsulin

In Figure 6.33, the percentage improvement compared to Voronoi is shown. It can be seen that the performance gap increases as more nodes are redistributed up to a certain point. After this point, the performance gap starts to shrink again. This comes back to our rationale for why we stopped testing at 50%. After a certain point, the network stops being a sparse-dense network and just becomes a dense network. The Voronoi implementation can again take advantage of this because of its way of distributing nodes, which is only concerned with distance to servers.

In Figure 6.34, the percentage improvement compared to Tsitsulin is shown. The difference is very noticeable. Not only is the performance considerably worse, but we can also notice a trend the

more nodes are redistributed. From a results perspective, this is not very noticeable. We already expected these results from our previous tests. These results prove that our implementation is much better suited to deal with randomness in general, which is likely because of the very different objective functions. Since Tsitsulin uses modularity calculations in their function, they are more susceptible to structural changes. Meanwhile, our implementation is mainly focused on proximity and load balancing, which are just indirectly related to the structure of the network.

Chapter 7

Conclusions and Future work

This thesis implemented an unsupervised Graph Neural Network approach for the problem of clustering IoT networks with regards to load balancing between clusters. This solution was rigorously tuned and tested under a multitude of conditions and topologies.

In IoT networks, most devices are running on batteries, which makes the environment very constrained from an energy perspective. One of the addressable factors that are directly responsible for energy consumption is the communication load. Load unbalance between clusters can lead to uneven consumption of energy, increased congestion and even gaps in coverage due to dead devices. Through our design, we cluster the network with a focus on balancing the load between the clusters, to have a more even spread of the load across the network, thus prolonging the lifetime of the network.

The experimental results have proven the viability of our design under a multitude of topologies and sizes. Of note is the fact that our implementation always performed well. Even if it sometimes was not the first performance-wise, the gap was always marginal, making our unsupervised GNN solution the best choice overall with respect to load balancing of the clusters in the network.

There certainly are cases where our solution is much better suited. The cases that leverage the GNN design, like networks with dynamic elements where the model is given time to learn the particularities of the network or well-structured networks with similar topologies and well-defined rules. Additionally, the case of variable density networks has proven ideal for our implementation. This is particularly good because these types of networks are actually fairly common in real-world scenarios. The higher density areas in the network can be points of interest like intersections in a city network or meeting rooms in a smart building. Depending on the dataset, our model was able to achieve over 50% improvement when compared to the baseline implementations.

The current design was tested with mainly simple use cases in a scenario with relatively lenient requirements. This is not always the case. It would provide an interesting avenue for further research to check the design under different scenarios, for example, one that has unusual patterns of communication. This way, the load balancing component would be put under pressure and see how well it performs.

Due to the nature of IoT networks, this is a field where we notice a high use for Reinforcement learning. The reasoning for this is simple, the simulation environment or the real-world environment of the network can be modelled as the environment component of the model. This would be an interesting avenue for future research since it would remove the limitation currently imposed through the loss function of having cluster cohesion.

Bibliography

[1] Anurag Verma, Surya Prakash, Vishal Srivastava, Anuj Kumar, and Subhas Chandra Mukhopadhyay. Sensing, controlling, and iot infrastructure in smart building: A review. *IEEE Sensors Journal*, 19(20):9036–9046, 2019. 1

[2] Ramachandra Gurunath, Mohit Agarwal, Abhrajeet Nandi, and Debabrata Samanta. An overview: Security issue in iot network. In *2018 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2018 2nd International Conference on*, pages 104–107, 2018. 1

[3] Francesca Meneghello, Matteo Calore, Daniel Zucchetto, Michele Polese, and Andrea Zanella. Iot: Internet of threats? a survey of practical security vulnerabilities in real iot devices. *IEEE Internet of Things Journal*, 6(5):8182–8201, 2019. 1

[4] Muhammad Shoaib Farooq, Shamyla Riaz, Adnan Abid, Tariq Umer, and Yousaf Bin Zikria. Role of iot technology in agriculture: A systematic literature review. *Electronics*, 9(2), 2020. 1

[5] Nitin B. Raut and N. M. Dhanya. A green dynamic internet of things (iot)-battery powered things aspect-survey. In Millie Pant, Tarun Kumar Sharma, Rajeev Arya, B.C. Sahana, and Hossein Zolfagharinia, editors, *Soft Computing: Theories and Applications*, pages 153–163, Singapore, 2020. Springer Singapore. 1

[6] Felisberto Pereira, Ricardo Correia, Pedro Pinho, Sérgio I. Lopes, and Nuno Borges Carvalho. Challenges in resource-constrained iot devices: Energy and communication as critical success factors for future iot deployment. *Sensors*, 20(22), 2020. 1

[7] Hassan Elahi, Khushboo Munir, Marco Eugeni, Sofiane Atek, and Paolo Gaudenzi. Energy harvesting towards self-powered iot devices. *Energies*, 13(21), 2020. 1

[8] Syed Yasmeen Shahdad, Asfia Sabahath, and Reshma Parveez. Architecture, issues and challenges of wireless mesh network. In *2016 International Conference on Communication and Signal Processing (ICCSP)*, pages 0557–0560, 2016. 1

[9] Pankaj Kumar Mishra and Shashi Kant Verma. A survey on clustering in wireless sensor network. In *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–5, 2020. 1

[10] J. Amutha, Sandeep Sharma, and Sanjay Kumar Sharma. Strategies based on various aspects of clustering in wireless sensor networks using classical, optimization and machine learning techniques: Review, taxonomy, research findings, challenges and future directions. *Computer Science Review*, 40:100376, 2021. 1, 2

[11] Behrouz Pourghebleh and Vahideh Hayyolalam. A comprehensive and systematic review of the load balancing mechanisms in the internet of things. *Cluster Computing*, 23(2):641–661, 2020. 1

[12] Wooi King Soo, Teck-Chaw Ling, Aung Htein Maw, and Su Thawda Win. Survey on load-balancing methods in 802.11 infrastructure mode wireless networks for improving quality of service. *ACM Comput. Surv.*, 51(2), feb 2018. 1

[13] Dipak Wajgi and Nileshsingh V Thakur. Load balancing algorithms in wireless sensor network: a survey. *IRACST International Journal of Computer Networks and Wireless Communications*, 2:2250–3501, 2012. 2

[14] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020. 3

[15] Weiwei Jiang. Graph-based deep learning for communication networks: A survey. *Computer Communications*, 185:40–54, 2022. 3

[16] Shiwen He, Shaowen Xiong, Yeyu Ou, Jian Zhang, Jiaheng Wang, Yongming Huang, and Yaoxue Zhang. An overview on the application of graph neural networks in wireless networks. *IEEE Open Journal of the Communications Society*, 2:2547–2565, 2021. 3

[17] Nosayba Al-Azzam and Ibrahem Shatnawi. Comparing supervised and semi-supervised machine learning models on diagnosing breast cancer. *Annals of Medicine and Surgery*, 62:53–64, 2021. 3

[18] Jan-Willem van Bloem, Roel Schiphorst, Taco Kluwer, and Cornelis H Slump. Spectrum utilization and congestion of ieee 802.11 networks in the 2.4 ghz ism band. *Journal of green engineering*, 2(4):401–430, 2012. 9

[19] Zachary Hays, Grant Richter, Stephen Berger, Charles Baylis, and Robert J Marks. Alleviating airport wifi congestion: An comparison of 2.4 ghz and 5 ghz wifi usage and capabilities. In *Texas Symposium on Wireless and Microwave Circuits and Systems*, pages 1–4. IEEE, 2014. 9

[20] Ejaz Ahmed, Ibrar Yaqoob, Ibrahim Abaker Targio Hashem, Imran Khan, Abdelmuttlib Ibrahim Abdalla Ahmed, Muhammad Imran, and Athanasios V Vasilakos. The role of big data analytics in internet of things. *Computer Networks*, 129:459–471, 2017. 12

[21] Iqbal Muhammad and Zhu Yan. Supervised machine learning approaches: A survey. *ICTACT Journal on Soft Computing*, 5(3), 2015. 12

[22] Yuji Roh, Geon Heo, and Steven Euijong Whang. A survey on data collection for machine learning: A big data - ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering*, 33(4):1328–1347, 2021. 13

[23] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018. 13

[24] Zhilin Yang, William Cohen, and Ruslan Salakhudinov. Revisiting semi-supervised learning with graph embeddings. In *International conference on machine learning*, pages 40–48. PMLR, 2016. 15

[25] Sergei Ivanov, Sergei Sviridov, and Evgeny Burnaev. Understanding isomorphism bias in graph data sets. *arXiv preprint arXiv:1910.12091*, 2019. 16

[26] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pages 593–607. Springer, 2018. 16

[27] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. 17, 19

[28] Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V Le. Self-training with noisy student improves imagenet classification. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10687–10698, 2020. 17

[29] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018. 17

[30] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. Graph contrastive learning with augmentations. *Advances in Neural Information Processing Systems*, 33:5812–5823, 2020. 17

[31] Zilong Huang, Xinggang Wang, Lichao Huang, Chang Huang, Yunchao Wei, and Wenyu Liu. Ccnet: Criss-cross attention for semantic segmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 603–612, 2019. 17

[32] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pages 639–648, 2020. 17

[33] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. 17, 18

[34] Petar Veličković, William Fedus, William L. Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. Deep graph infomax, 2018. 18, 19

[35] Yanqiao Zhu, Yichen Xu, Feng Yu, Qiang Liu, Shu Wu, and Liang Wang. Deep graph contrastive representation learning. *arXiv preprint arXiv:2006.04131*, 2020. 18

[36] F Dehne and H Noltemeier. Voronoi trees and clustering problems. *Information Systems*, 12(2):171–175, 1987. 18

[37] Anton Tsitsulin, John Palowitch, Bryan Perozzi, and Emmanuel Müller. Graph clustering with graph neural networks. *arXiv preprint arXiv:2006.16904*, 2020. 18

[38] Yanqiao Zhu, Yichen Xu, Feng Yu, Shu Wu, and Liang Wang. Cagnn: Cluster-aware graph neural networks for unsupervised graph representation learning. *arXiv preprint arXiv:2009.01674*, 2020. 19

[39] Filippo Maria Bianchi, Daniele Grattarola, and Cesare Alippi. Spectral clustering with graph neural networks for graph pooling. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 874–883. PMLR, 13–18 Jul 2020. 19

[40] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000. 19

[41] Sambaran Bandyopadhyay and Vishal Peter. Unsupervised constrained community detection via self-expressive graph neural network. In Cassio de Campos and Marloes H. Maathuis, editors, *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, volume 161 of *Proceedings of Machine Learning Research*, pages 1078–1088. PMLR, 27–30 Jul 2021. 19

[42] Ehsan Elhamifar and René Vidal. Sparse subspace clustering: Algorithm, theory, and applications. *IEEE transactions on pattern analysis and machine intelligence*, 35(11):2765–2781, 2013. 19

[43] Abdullah Khanfor, Amal Nammouchi, Hakim Ghazzai, Ye Yang, Mohammad R. Haider, and Yehia Massoud. Graph neural networks-based clustering for social internet of things. In *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1056–1059, 2020. 19

[44] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical report, Stanford, 2006. 19

[45] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996. 19

[46] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):5363–5370, Apr. 2020. 20

[47] Yuhang Yao and Carlee Joe-Wong. Interpretable clustering on dynamic graphs with recurrent graph neural networks. In *AAAI*, pages 4608–4616, 2021. 20

[48] Qingzhi Liu, Tiancong Xia, Long Cheng, Merijn van Eijk, Tanir Ozcelebi, and Ying Mao. Deep reinforcement learning for load-balancing aware network control in iot edge systems. *IEEE Transactions on Parallel and Distributed Systems*, 33(6):1491–1502, 2022. 20

[49] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. 20

[50] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 28

[51] Joakim Skarding, Bogdan Gabrys, and Katarzyna Musial. Foundations and modeling of dynamic networks using dynamic graph neural networks: A survey. *IEEE Access*, 9:79143–79168, 2021. 41