uc3m | Universidad Carlos III de Madrid

University Degree in Telematics Engineering
Academic Year 2022-2023

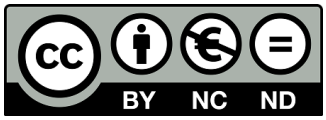
Bachelor Thesis

Graph Neural Networks as Digital Twin

Alejandro Calvillo Fernandez

Jorge Martín Pérez

Leganés, 20^{th} of February 2023



ABSTRACT

Over the last few years, Machine Learning has become an essential part within the deployment of some services. New state-of-the-art telecommunications technologies will come with some of these models implemented natively that will help to perform tasks such as service orchestration, packet flow prediction, or even with cybersecurity issues. With the rapid increase of traffic we are experimenting year after year, new 3GPP technologies are not enough to face by themselves the challenges of new times. For this reason, the search for a solution to these problems in models based on artificial intelligences is an open field in the world of research.

Graphs Neural Networks (GNNs) are a very promising field of research for solving these types of issues. GNNs are Neural Networks that operate directly with the graph structure, enabling them to learn directly from the nodes, edges, and the topology, in order to seek for more complex relations regarding other data structures. Learning directly from the structure of a molecule, the relations between humans, or the computer networks could solve many issues in the area of interest. Along this thesis, we will learn how this machine learning models works internally, and we will develop a simple model. This study aims to provide for every researcher, student, or other people with interest in the subject, with basic knowledge in order to continue exploring this field.

Keywords: Graph Neural Networks, Graph Convolutional Networks, Machine Learning, Digital Twin.

DEDICATORIA

A todas las personas implicadas en el desarrollo de este proyecto. En especial, a Jorge, por su paciencia y vocación por la docencia.

A mi familia más cercana. Susana, Jose, Lucía y Nacho. Por seguirme en este largo camino, brindándome el apoyo necesario para vivir una vida maravillosa a vuestro lado.

A Cristina, por enseñarme que un hogar es aquel lugar donde estemos juntos.

A Jimmy y Alberto, porque allá donde estemos, que sigamos siendo un único "Cerebro Galaxia".

A Irene, que tu valentía y fortaleza te lleven tan lejos como te propongas llegar.

A Chema y Alvaro, por hacer los días de "Los Últimos del Sabatini" tan especiales.

A David y Lucía, por compartir tantas noches de frustración y días de alegría durante todo este tiempo.

A todos ellos, os merecéis todo lo bueno que os pase.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Since the dawn of philosophy, humans, as a specie, have tried to understand nature by their senses, using our rational thinking as a tool. Over time, humanity has advanced in the understanding of natural phenomena and developed new technologies to improve the quality of life.

Despite the modern times we live in, and the modern inquietudes we have now, trying to understand nature and the quest for learning how to predict it are something we share with the ancient Greeks.

In the 21st century, one of the areas of greatest progress and development is **artificial intelligence** and, in particular, machine learning. This field is based on the idea that machines can learn from data without being explicitly programmed, allowing them to perform complex tasks and make decisions based on patterns and relationships in the data that humans cannot see at a simple glance.

Since the start of the development of **Machine Learning**, this technology has been successfully applied in a wide variety of fields, from medicine and biology to finance and robotics, and now to graphs. For example, it is being used to develop more precise medical diagnostic systems, to improve the efficiency of online recommendation systems, and to control robots and autonomous vehicles, among many other applications in today's society.

In the case of graphs, we have not to gone so far from Ancient Greece. Trying to represent the relationships between natural phenomena is part of the essence of humanity. In graph theory, a **Graph** is a representation with nodes and links that represents objects and the relations between them. Graphs have been used in infinite science areas to represent relations and patterns between data.

In this era, graphs, have become an essential part of data visualization. From attempts at understanding the iterations within a social network, why molecules produce bad odor or why traffic jams occur, graphs are a crucial part of, not only for the visualization of these issues, but now, also processing them in order to make predictions to solve them.

Closer to my area of knowledge, in telecommunications, graphs are useful to represent computer networks. If we digitalize this representation, we come with a **Digital Twin**. A Digital Twin is a precise digital model of a real object or system that helps us understand its behavior and to perform tasks with the provided information.

The idea of making predictions around a digital twin using graphs and machine learning is the main motivation for the **Graph Neural Network** (GNN) theory. This technology born with the motivation to merge these two areas. The idea around this field is to apply *Neural Networks*[1], a technology within machine learning known as *deep learning*,

to the case of graphs, in order to achieve knowledge between complex relations.

In this study, we are going to dig deeply into how this technology works. We will understand how can we perform complex mathematical operations, such as the *convolution* of graphs, and we will learn how to use data we have obtained from *Universitat Politècnica de Catalunya Graph Neural Networking challenge 2022*[2], adapt it and, in the end, develop our first GNN model.

After reading this thesis, you will be prepared to understand how useful this technology is to modern ages and where we can apply it.

1.1. Motivation and objective

The main motivation of this project is to bring closer machine learning in telecommunications in the case of computer networks. Two worlds that are not so far, but we consider will be closer within the passing years.

For this task, we have decided to make a study on GNN. Part of the time devoted to this thesis has been spent in finding, understanding and putting together the necessary mathematical background. This has been an effort motivated by the intention to bring this knowledge to the reader in an accessible and understandable approach. In order to do so, all computations will be deeply stated step by step for the reader to follow.

Once all mathematics are understood, we will develop a simple model using **Python** as a coding language. This model will be trained with real digital twin data, and we will demonstrate that our machine learning is, in fact, learning.

In addition to the technological and scientific reasons, the personal aim for doing this project is motivated by the curiosity of getting machine learning closer to computer networks. Many new technologies, such as 6G, will native implement machine learning in many tasks for improving performance [3]. I am moved by the idea that GNNs can optimize many of these processes, in order to make next-generation technologies more reliable.

1.2. Structure of the study

In an attempt to clarify the structure of this thesis, we will proceed with a brief summary of each chapter of this work:

- **Introduction**: An explanation of the project in order to get closer with the reader.
- **State of the Art**: description of where are we today in machine learning features and where are applied in telecommunications.
- **Graph Neural Networks**: introduction to graphs, and first steps into GNNs.

- **Mathematical Context of GCNs**: study of how the convolution works for the case of graphs, and introduction to the reader to some GNNs that use the convolution, known as *Graph Convolutional Networks* (GCN)
- **Implementation of a GCN**: development of a simple model using all knowledge we have acquired.
- **Estimation and Planning**: estimation of the time spend on computations, writings, and development of this thesis.
- **Conclusion**: discussion about the utility and services of GNN.
- **Bibliography**: all sources of information used to overcome this study.

2. STATE OF THE ART

2.1. Machine Learning Branches

From a global point of view, this study is about Machine Learning (ML). Machine Learning involves the process of learning from data that software applications have to follow, by learning from data acquired in previous processes, in order to improve their accuracy in making predictions or decisions[4]. This idea is motivated by the notion of intelligent machines being able to perform computations and learn from previous states. When we think about ML, we are suggested that this idea is a recent concept. However, in [1], we can see that it comes from long ago. More precisely, the first Neural Networks (NN) proposed date from 1943.

Nowadays, the main goal in ML is to discover relations between input and output data. Among these relations, we can have three different types of models; *classification, regression* and *structured models*:

- **Classification models**: this type of learning models are used to forecast group membership for data instances [5].
- **Regression models**: in this case, regression analysis refers to the method of studying the relationship between independent variable and dependent variable[6].
- **Structured models**: Structured machine learning is the process of extracting structured proposals from richly structured data, used in many fields, such as in natural language processing, [7].

In addition, we can classify ML by the type of learning they use to perform this metrics in: *supervised, unsupervised* and *reinforcement learning* [8]. We think it is important, for the fully understanding of this study, to make a brief explanation for each of them, which we will do in the following sections.

2.2. Supervised Learning

In supervised learning, we have an input associated to a labeled output. During the training, the ML model aims to produce output matching the labeled output. In the testing phase, we recreate the same process with a set of expected outputs [9]. This ML models have been extensively used in many fields such as object recognition [10], medical diagnosis [11] or stock prices prediction [12]. In this section, we will present the reader three different models that include this type of learning process:

- **K-Nearest Neighbor** (KNN) [13]: KNN works measuring the distance between different features value. With that distance, we can group within the space with the same feature. All nodes inside this space are classified into the same category. This method is commonly used in ML due to suitability for multi-class problems.
- **Random Forest** [14]: consists of multiple decision trees. A decision tree is a representation where each "leaf" represents a class and each branch represents a coincidence of features for the data. In the case of random forest, a random subset of features is selected in order to avoid over-fitting.
- **Support Vector Machine** (SVM)[15]: algorithm that learns by example to assign labels to objects, in order to locate a hyperplane in the feature space and determine the greatest margin between several classes.

2.3. Unsupervised Learning

In [16], we find a really accurate definition for Unsupervised Learning: "in contrast with supervised learning, there are no explicit target outputs or environmental evaluations associated with each input; rather, the unsupervised learner brings to bear prior biases as to what aspects of the structure of the input should be captured in the output". In short, in Unsupervised Learning, we try to classify different samples of data based on the similitude with the input features.

Typically, unsupervised learning problems are formulated to be resolved with a *Markov Decision Problem* (MDP). This model tries to approximate the problem to a *Markovian* solution. Although this implementation is commonly used to solve unsupervised problems, reinforcement learning plays a crucial role in MDPs [17].

2.4. Reinforcement Learning

Reinforcement Learning (RL) tries to make the learning process without structured data, by trial and error[18]. Similar to the human brain, RL has a higher level understanding of the visual world through actions and feedback, which makes it suitable to take real-world decisions. This allows us to make fully autonomous agents that interact with the environment[19].

This autonomous agent interacts with the environment by observing a state and making a decision. When an action is taken, another state is passed to the agent. The learning process is ensured by rewards. In each action performed by the agent, there is a reward as feedback from the environment. With this reward, learning a control technique that maximizes the expected return is the agent's aim. For that reason, every interaction the agent has with the environment produces information that it utilizes to update its understanding.

2.5. Graph Neural Networks studies

Although all mentioned ML methods are good for structure data, modern requirements postulate other types of data, such as graphs (for expressing social networks relations, citations chains, molecules representation...). Neural Networks, used to solve supervised/unsupervised problems, do not have sufficient information about the relation between nodes that produce inputs. That means, NN do not have any information of the neighborhood of a node. Graph Neural Networks (GNNs) are able to capture, and process, the topology of how nodes are connected between them, in order to manage social networks relations, citation chains or molecules representation. [20].

In [21], Elan Markowitz et al. proposed the use of tensorial functions in order to make it easier to handle with nodes and the relations between graphs, but also the treatment with the data features.

In [22], a new technique is introduced by S.S. Du et al. They proposed to use a *kernel* function to measure similitude between two graphs and then learn this *kernel* by using *Neural Networks*.

Another exaple of a recent study on this area is in [23], where K.Xu et al. introduces a new technique, what they called *"Jumping Knowledge Networks"*. This technique, permits the neural network to represent more precisely the information stored in the nodes.

Petar Velickovic et al., in [24] proposed a new idea motivated by the usage of *Neural Networks* for computing task inside the graph, such as shortest-path or identification of connected components.

Graph Neural Network is a current object of study in many institutions; new content is constantly being released. This is what powered the motivation of this study; making it easier for every researcher, student or inquisitive individuals: bringing people closer to the GNNs world.

3. GRAPH NEURAL NETWORKS

In order to explain more complex concepts, as GNN models can be, we need to go to the basis of graphs.

First, we need to know what a graph is, what kind of data can be naturally phrased as a graph and what makes graphs special from other types of data.

From this point we can start by defining what a GNN is, taking a look into the mathematics of this model, and move gradually from bare-bones implementation to modern models architectures.

To help us sort through this knowledge, we will use "A Gentle Introduction to Graph Neural Networks" [25] by Google Research's team as principal article and Clara Grima's book "En Busca del Grafo Perdido" [26] as a support source of knowledge.

3.1. A brief introduction to Graphs

3.1.1. Definition of Graph and where to find them

According to the article published in Distill [25], a graph represents the relation between a collection of entities. By this relation, we can join entities (nodes) by links (edges).

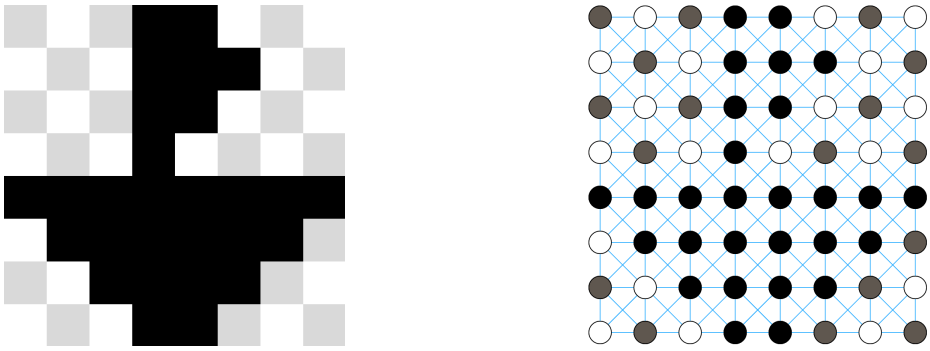
In all these components we can store information, for example:

- In **Edges** we can store edge identities or edge weights.
- In **Nodes** we can store the node identity or number of neighbors.
- In the **Global** we can store the number of nodes or the shortest path.

Now that we know what a graph is, we can get into where we can find them, or what type of data can we represent with them.

In the first place, we have **Images**. If we think of an Image as an NxM matrix, identifying each cell as an independent pixel, we can also think of it as a graph, where each node represents a pixel, connected by edges to adjacent pixels. Each non-border vertex has 8 neighbors with a 3-dimensional vector that represents its RGB *(R, G, B)* code of it.

Below there are pictured two images showing the same data. Picture on the left (see Fig. 3.1a) is an 8×8 matrix with a simple sketch of a boat. The picture on the right (see Fig. 3.1b) shows us the same data but in a graph, identifying each node with a color.



(a) Boat Draw (b) Boat Graph

Fig. 3.1. Two images generated using https://www.pixilart.com/ for (a) and https://virtual-graph-paper.com/ for (b)

Following with the second point, we have **Text**. Text is composed of sentences with a meaning by themselves. A sentence in a language is an ordered sequence of words. If we also see words as tokens, we can represent it by a directed graph, in English, from left to right. We will see this with an easy example:

Suppose we have *"Graphs are great"* as data. Words now will be tokens to identify nodes, and edges will show a new interesting property: they can be undirected or directed (see Fig. 3.2).

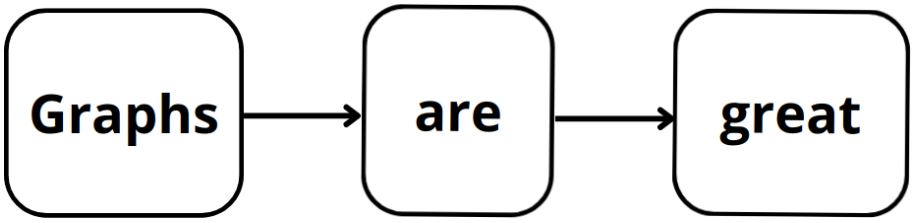


Fig. 3.2. An easy example of how to represent text with graphs

As we defined, text is a sequence of words (tokens), so we need edges to be directed in one direction, that's why we represent them as arrows. In other types of data, directed edges can point not only to the next node, but also to themselves.

This new representation that we have just learned of text as graphs can be really useful if we want to digitalize languages for future treatment, for example, to model predictive keyboards.

In reality, texts, and images are not encoded like that. If we take a look, graphs made from this type of data are redundant but will also follow really regular structures.

3.1.2. Graphs as representation of nature

Now, we will dig into types of data that can be more interesting to study them with a graph. Intuitively, it is easier to liken natural structures to graphs than to any other similar data representative structure.

This can be the case of **Molecules**. Molecules are complex structures where atoms are bonded to each other's. When two of these atoms join together, a covalent bold appears, joining the atoms but maintaining a certain distance. Now we have, on one hand, atoms that will be identifying our nodes and, on the other hand, covalent bonds, that will be the edges of the graph.

In Fig. 3.3a, we see a simple representation of a *Benzene* molecule, made of a central structure of *carbon* atoms in a structure of a pentagon, with one atom of *hydrogen* joined to each corner. In Fig. 3.3b, we can see the graph that symbolizes this molecule. Now we can observe that it has the same structure as benzene, black nodes standing for *carbons* and *hydrogens* being white.

(a) Benceno (b) Benceno Graph

Fig. 3.3. Image taken from https://free3d.com/ for (a) and generated using https://virtual-graph-paper.com/ for (b)

It is more accurate to represent nature with this type of graph than with an image graph. Digitalizing molecules will make it easier to study and work with them. Graphs are a powerful weapon for this task, as they are quite easy to compute in this role. Now, we will see other examples where graphs can represent more ambiguous concepts, such as **Human Relations**.

We could take a look at **Social Networks** as a powerful tool to study *human relations*. We can find patterns in society by studying the relationships between two or more individuals. A node can identify a person, and the edges can represent the union between them and those in their environment. We can use this tool to study societal structures in order to, for example, predict people's behavior patterns.

Patterns into society can normally being discovered by relation between two or more

individuals. As nodes can identify persons, and edges can be the relation between them, we can use this tool to study society structures in other, for example, to predict people conductive patterns.

But we have not talked yet about the object of this study. Computer Network graphs, being nodes a representation of any entity or component (identifying correctly all of them) and edges standing as links, are the basis for **Digital Twins**, that will be studied in following chapters.

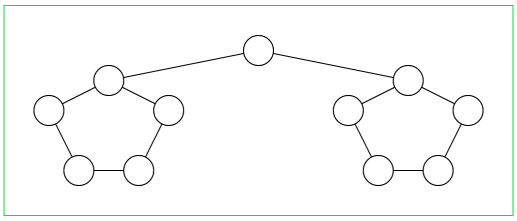
Not only representing or storing information, but predicting data can be done by using graph models, as we will see in the next section.

3.1.3. Use of graphs for different tasks

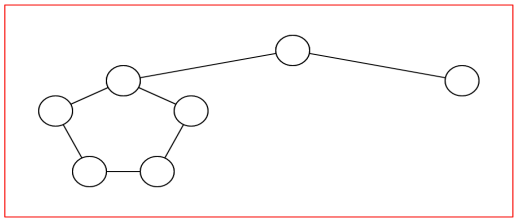
At this point, we have seen a couple of examples where we can represent nature with a graph. Now, with all the data that this tool can provide for us, what we want to do is to perform prediction tasks. We can work with three general tasks: **Graph-level, Node-level and Edge-level**

- **Graph-level task:** The goal of this prediction type is to obtain a property of an entire graph. Let's see this with an example:

  Suppose we want to identify a molecule that has two rings. As we can see, if our model is correctly trained, molecules with exactly two rings will be marked as correct (see Fig. 3.4a), but any other will be marked as wrong (see Fig. 3.4b).



(a) Two rings



(b) One ring

Fig. 3.4. Images generated using https://virtual-graph-paper.com/

  There are more complex examples and theories, for example, images classification with MNIST and CIFAR, which are not objects of this study.

- **Node-level task:** the goal is to predict the identity of each node within a graph. In Clara Grima's book [26], we can read about a jigsaw called "El Guateque". We will expose it below:

  Alicia and Blas are a couple who has met up with other four couples to have dinner. When they arrive, everybody gets into the restaurant with their partners and greet all the other attendees, by giving a handshake or two kisses (as is the Spanish

tradition). Later in the dinner, when everybody is finished with their desserts, Blas suggests all participants to write on a piece of paper how many people they have greeted with a handshake. They all do so, and after that, they give the paper to Blas, which mixes them all. By chance, all numbers written are different, there is no number of handshake repeat. The question to answer is: How many people does Alicia greet with a handshake?

The first hint we need to think about is that the 9 answers are different, so the ten possible responses are {0,1,2,3,4,5,6,7,8,9}. From here, we can discard number 9, because nobody greets their partner in the moment of meeting for dinner with the other couples. By now, the set of numbers we have is {0,1,2,3,4,5,6,7,8}. From here we can take graphs to solve the problem.

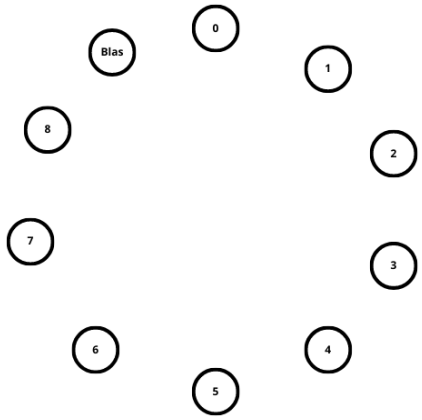
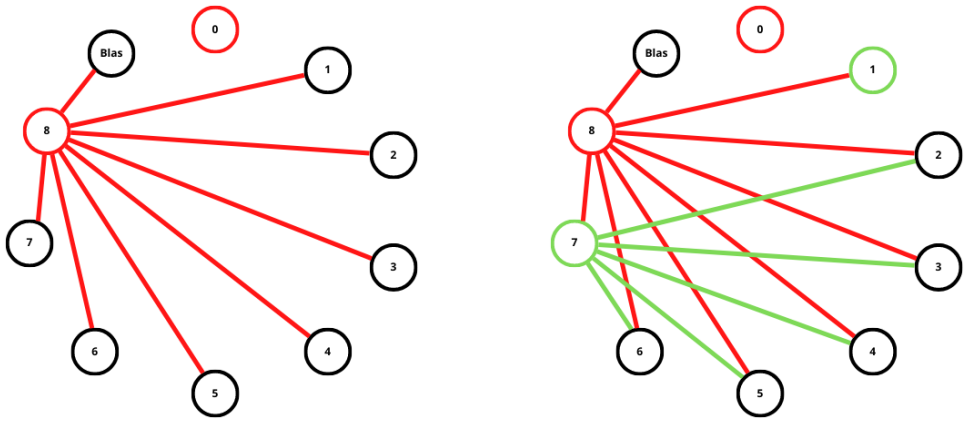


Fig. 3.5. All nodes numbered

We name each node by the number of handshakes to predict which node is Alicia. Now, we can start joining nodes. The person who greets handshaking eight people gives a handshake to everybody except the one who didn't handshake anyone, in this case node "0" (see Fig. 3.6a). Then, what we can determine is that node "8" and node "0" are a couple.

We can continue with the next one. Node "7" greets everyone, but node "1", with a handshake (notice that node "1", after the first step, cannot greet anyone else by a handshake because they already greeted node "8"). That means node "7" and "1" are another couple (see Fig. 3.6b).



(a) Eight finds his partner

(b) Seven finds his partner

Fig. 3.6. Identifying couples with graphs

If we do the same with all the remaining nodes, we can identify Alicia as node "4". Answering the question, Alicia greets four people with a handshake. Following mathematical algorithms we can, not only identify nodes, but predict identities, as we have just done.

- **Edge-level task:** the goal is to predict the relationships between each node. Deep-learning models can be used to predict this connection between entities.

  In an image, for example, we can represent each concerning object as a node, and the relations between them with edges.

We will see that GNN can solve all the three prediction types explained above, at the same time.

In this study, we will be focused on **Graph Convolutional Networks(GCN)**. We will study what is happening inside a GCN, and we will try to code a simple model to test how it works.

4. MATHEMATICAL CONTEXT OF GCNS

Before we can continue with the definition of what GCNs are, we need some mathematical context to understand the computations this architecture makes.

4.1. Convolutional Networks

Convolutional Networks (ConvNets) are powerful architectures to solve high-dimensional learning problems[27], led by one main assumption: all data is compositional.

This data is composed of patterns that are **Local**, **Stationary (Shared Patterns)** and **Hierarchical (Multiscale)**.

For example, as we can see in previous sections, images can be decomposed into a Euclidean domain (grid). Each node of the graph, taking every pixel as a node, contains information of the RGB component of it as:

$$f = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \in \mathbb{R}^3 \tag{4.1}$$

We can apply the same intuition to graphs, in the following way:

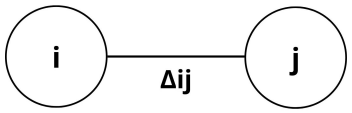


Fig. 4.1. A simple example of two nodes and one edge

Suppose this example represents two neurons. Let f be the function representing the number of ions due to the Nernst Potential [28]:

- $Neuron_i : f = \begin{pmatrix} K^+ \\ Mg^{2+} \\ Ca^{2+} \\ Na^+ \\ Cl^- \end{pmatrix}_i \in \mathbb{R}^d$
- Being this edge, the $neuron_i$ axon: $\Delta_{ij} = \begin{cases} 1, & \text{if } i \text{ transmit} \\ 0, & \text{otherwise} \end{cases}$
- $Neuron_j : f = \begin{pmatrix} K^+ \\ Mg^{2+} \\ Ca^{2+} \\ Na^+ \\ Cl^- \end{pmatrix}_j \in \mathbb{R}^d$

But yet, we have not discussed at what point we use convolutions. This will be further explained in the following section.

4.1.1. Convolutional layer

The goal of this following sections is to redefine convolution for the case of graphs. By now, we have discovered two ways: by the **Spatial Convolution** or by the **Spectral Convolution**.

First, it is important to define the concept of Kernel. A Kernel is an algorithm in dot product space that helps us into pattern analysis [29],

For now, let's take the example of an image as a grid in a Euclidean domain, such as a hand-drawn person.

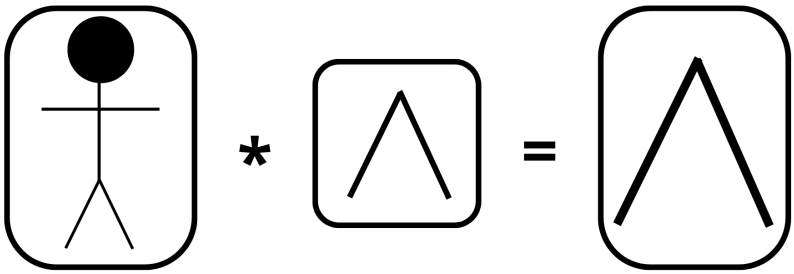


Fig. 4.2. Example of image processing using kernel

In mathematical language, this example can be written as:

$$f^{l+1} = w^l * f^l \tag{4.2}$$

which corresponds to the definition of the **Convolutional Layer**. Where f is the image of a function vector of size $n_1 n_2 d$ (n_1 is the number of pixels on x-axis, n_2 is the number of pixels of y-axis and d is the dimensions of the pixel), w is the kernel with another size, for example $3 \times 3 \times$ d, and l is the layer where we are.

By the equation exposed above (4.2), we can use the kernel as a filter to apply to the image, so we can select the part of information we want. But how can it be done? This takes us directly to defining the **convolution as a template matching**:

$$f^{l+1} = w^l * f^l = \sum_{j \in N_i} < w^l_{ij}, f^l_{i-j} > = \sum_{j \in N_i} < w^l_{ij}, f^l_{ij} > = \sum_{j \in N_i} < [w^l_{ij}, f^l_{ij}] > \tag{4.3}$$

Using the convolution, we will compare ordered nodes, always with the same position. In other words, if we think of the first example (See Fig. 4.2), that means we will compare ${}_j3$ with ${}_j3$ if we visualize it as an ordered grid. As we can see, we do the summation between the neighbors to apply the convolution as the summation of a vectorial product. Notice the simplification of $i - j$ as ij, this is because we flip up and down, and left and right, so it does not change anything.

This statement, leads us directly to a limitation: What about unordered graphs? Can we extend template matching to network graphs?

To solve this, we need to define the **Fourier transform of the convolution** as the pointwise product of their Fourier transforms:

$$\mathscr{F}(w * f) = \mathscr{F}(w) \odot \mathscr{F}(f) \rightarrow w * f = \mathscr{F}^{-1}(\mathscr{F}(w) \odot \mathscr{F}(f)) \tag{4.4}$$

The motivation for using the Fourier transform is because the convolution in the frequency domain, it is just a simple multiplication.

To make the Fourier Transform in the case of graphs with unordered nodes, we will need to redefine the convolution. This will be done in the following section, defining the **Spatial** and **Spectral Convolution**.

4.2. Spatial Convolution

The main motivation to redefine the convolution is the absence of node positioning. We can only take the index of the nodes, but this is not enough information to compare between the nodes. We can name a node with number 3, but this node can be the node 150. As we said in the previous section, we necessarily need to redefine the convolution as a **Template Matching** (4.3) for the case of graphs.

The simplest solution for this issue is to have one template vector to do the matching. This filter will keep invariance with the parametrization of the graph.

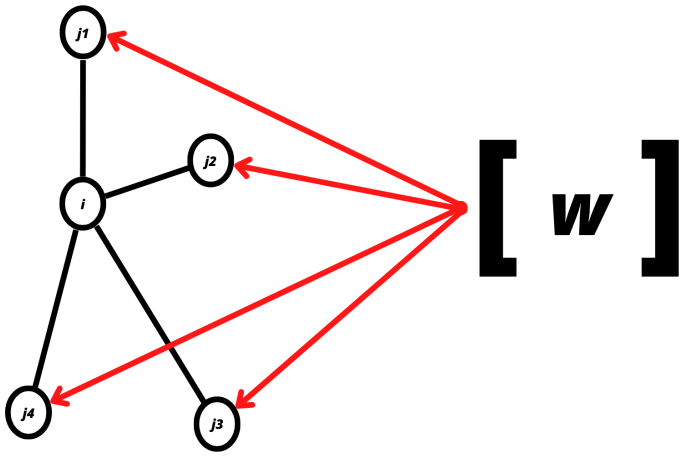


Fig. 4.3. Same filter invariant due to parametrization of the graph

As we see in Figure 4.3, we will have one filter n for the graph. Each node will have, $f_1^l, f_2^l, f_3^l, ..., f_n^l$, where f are the features of the node, n is the number of nodes in the graph (or partial graph) we want to take information and superscript l can be denoted as the l^{th} layer of the convolution.

Getting into the mathematics of the issue more in depth, we can denote the following expression as:

$$f_i^{l+1} = \eta(\sum_{j \in N_i} W^l f_{ij}^l) \tag{4.5}$$

Where f is a $dx1$ and W is a dxd matrix, respectively being d the number of features of the nodes and η is the activation function for the neuronal network (like ReLu, tanh...).

This way to do the **Template Matching** makes the filter Isotropic. Let's explain this in more detail.

All **Standard Convolutional Networks** produce **Anisotropic** filters due to usage of Euclidean grids to compute the filter. We can know where is up, down, left, and right.

In **Graphs Convolutional Networks** due to the absence of order in the nodes, there is no notion of space dimension to compute the convolution, that's why they produce **Isotropic** filters.

There is one way to get anisotropic filters on GCN, as we will see in futures sections.

4.2.1. Isotropic GNNs

VanillaGCN

The simplest implementation of a **Spatial GCN** is the**VanillaGCN** [30][31] [32]. What this GCN does is to compute the mean value over the neighbors. Thanks to that, this computation will be completely unconstrained by node ordering. So if we change the index, there will be no change in the computation.

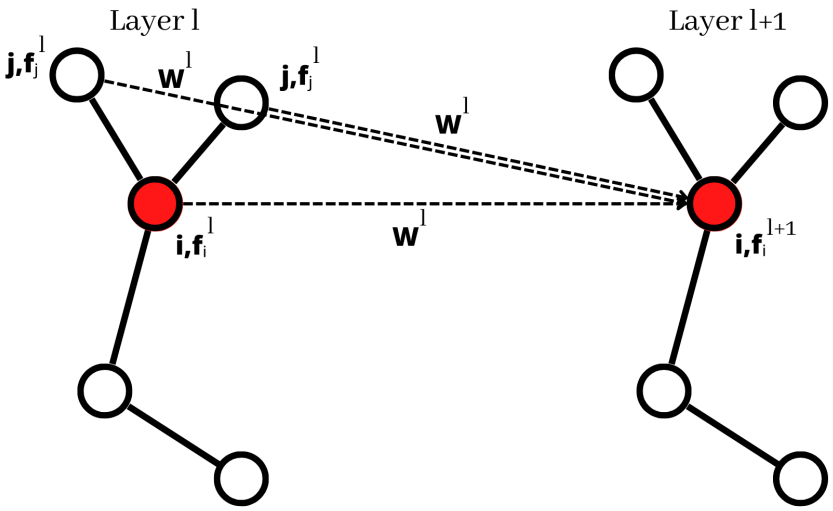


Fig. 4.4. Vanilla Spatial Template applying the mean to all nodes

With this method, we will use the same W whatever position we are in the graph. This sentence, drives directly to other intuition: we can compute the convolution independently of the graph size.

$$f_i^{l+1} = F_{GCN}(f_i^l, \{f_j^l : j \to i\}) \tag{4.6}$$

What this equation means is that the activation of the next layer $l + 1$, is an activation function, F_{GCN}, of the current layer on the node i(Represented in red in fig. 4.4) and the neighbourhood of the node i, $\{f_j^l : j \to i\}$.

4.2.2. Anisotropic GCNs

But what if we want some features that are stored in the edges (for example the bond in molecules, or losses in the link). How can we get Anisotropy in GCN? Let's see some solutions.

Firstly, we want this parametrization to be independently of the node's parametrization.

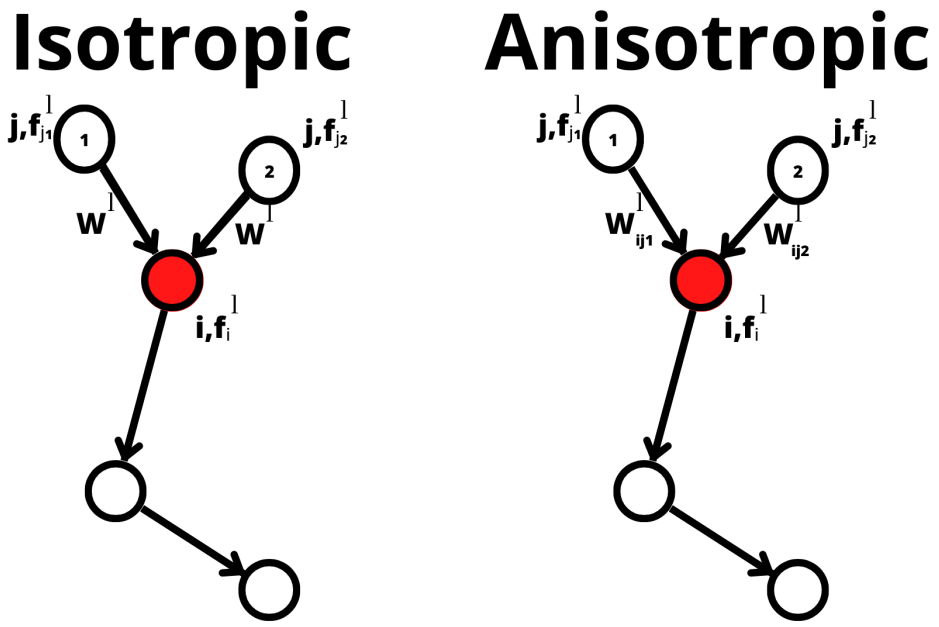


Fig. 4.5. Central node graph as example of the Laplacian. Where $W^l \neq W^l_{ij1} \neq W^l_{ij2}$

There are many models that implement anisotropy in GCN. In this study, we will define three of them.

MoNets[33] was the first model to deal with anisotropy. The main idea of MoNets is to use *Bayesian Gaussian Mixture Model*[34]. In short, with this model, we will learn the parameters by using the degree of the graph.

Graph Attention Networks[35] apply a *Soft-max* function around the neighborhood. By this, you can make some nodes more important than others.

Gated Graph ConvNets[36] model makes edge features explicit. If we want to make predictions in the edges, this explicity on the edges is important.

At this point, we have seen how to make the **Spatial Convolution** for the case of graphs. We have seen many models that implement this way to do the **Template Matching**.

Now, we will follow computing the Spectral Convolution. This new section will give the rest of the mathematical context to deeply understand what is going on deeply in a GCN.

4.3. Spectral Convolution

All these computations were made before by *Fang Chung*, in the book *Spectral Graph Theory* [37].

We will start defining the **Laplacian for the graphs**. The Laplacian will be used as a measure of smoothness. Respective to a node, we will compute the difference between the local value f_i and it's neighbor's average values as $(\Delta f)_i = f_i - \frac{1}{d_i}\sum_{j\in N_i} A_{ij} f_j$ where d is the degree of the selected node (number of neighbors). Fig. 4.6

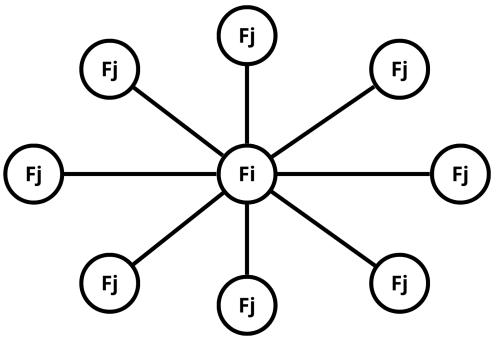


Fig. 4.6. Central node graph as example of the Laplacian

From now on, we will use the *Normalized Laplacian* equation as core:

$$\Delta_{nxn} = I - D^{-1/2} A D^{-1/2} \tag{4.7}$$

Where:

- I is the Identity Matrix
- $D_{nxn} = diag(\sum_{j\neq i} A_{ij})$
- A is the Adjacency Matrix, this represents connections between nodes. We will see this in an example below (See Fig. 4.7 and (4.13))

Let's also define the **Eigen-descomposition for graph Laplacian**:

$$\Delta_{nxn} = \Phi^T \Lambda \Phi \tag{4.8}$$

Where:

- Φ is the *Normalized Eigenvector matrix* as:

$$\Phi_{nxn} = \begin{bmatrix} \phi_1, & \cdots, & \phi_n \end{bmatrix} = \begin{bmatrix} | & & | \\ \phi_1 & \cdots, & \phi_n \\ | & & | \end{bmatrix} \tag{4.9}$$

And $\Phi^T\Phi = I, \langle \Phi_k, \Phi_{k'} \rangle = \delta_{kk'}$ as they are orthonormal basis.

- Λ is the *Diagonal Matrix of Eigenvalues*

$$\Lambda_{nxn} = diag(\lambda_1, \cdots, \lambda_n) = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_n \end{bmatrix}, 0 \leqslant \lambda_{max} \leqslant 2 \tag{4.10}$$

When we do the Eigen-descomposition we are going to factorize our Laplacian matrix into three matrices. Φ will contain the Laplacian Eigenvectors. These Eigenvectors, are called **Fourier functions**. On the other hand, if we plot all the Eigenvalues, Λ will be called the **Spectrum of the graph**.

At this point, we are ready to define **Fourier Transform for graphs**. What we are going to do to get the **Fourier series** is to decompose f using **Fourier functions** [38].

$$f_{nx1} = \sum_{k=1}^{n} \langle \phi_k, f \rangle \phi_k = \sum_{k=1}^{n} \hat{f}_k \phi_k = \Phi \hat{f} = \mathscr{F}^{-1}(\hat{f}) \tag{4.11}$$

Notice the step $\langle \phi_k, f \rangle = \hat{f}_k$ (both scalar). What we are doing is to project the Fourier functions into our function, so it give us the **Fourier transform**:

- So if we project Φ^T over f we will get the Fourier transform as: $\mathscr{F}_{nx1}(f) = \Phi^T f = \hat{f}$. We have defined the Fourier transform as a multiplication of matrices.
- If we do the **Inverse Fourier transform**: $\mathscr{F}^{-1}(\hat{f}) = \Phi \hat{f} = \Phi\Phi^T f = f$ (as $\Phi\Phi^T = I$ because they are Orthonormal basis).

Remember, our goal is to redefine convolution for the case of graphs. From the equation (4.4):

$$\begin{gathered} w * f = \mathscr{F}^{-1}(\mathscr{F}(w) \odot \mathscr{F}(f)) = \Phi_{nxn}(\hat{w}_{nx1} \odot \Phi^T f_{nx1}) = \\ \Phi(\hat{w}(\Lambda)_{nxn} \Phi^T f_{nx1}) = \hat{w}(\Phi^T \Lambda \Phi) f = \hat{w}(\Delta)_{nxn} f_{nx1} \end{gathered} \tag{4.12}$$

We know that when we apply a function of a vector on the eigenvalues, if we have some orthogonal basis, we can put it inside as: $\Phi(\hat{w}(\Lambda)_{nxn} \Phi^T h_{nx1}) = \Phi \hat{w}(\Lambda)_{nxn} \Phi^T h_{nx1} = \hat{w}(\Phi^T \Lambda \Phi) f$.

Also, $\hat{w}(\Lambda)$ can be described as: $\hat{w}(\Lambda) = diag(\Phi^T w) = \begin{bmatrix} (\Phi^T w)_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & (\Phi^T w)_n \end{bmatrix}$

This **Spectral Convolution** as $\hat{w}(\Delta) f$ has a really expensive computation. Φ is a full matrix, which contains n Fourier functions that are not 0, so we need to pay the price of $O(n^2)$ complexity (we will talk about this expression in the following sections).

Numerical Computation Example

But before jumping onto the next section, and to handle all this computation, we will perform the needed calculations by following this simple example exposed below.

Suppose we have the following graph (See Fig. 4.7):

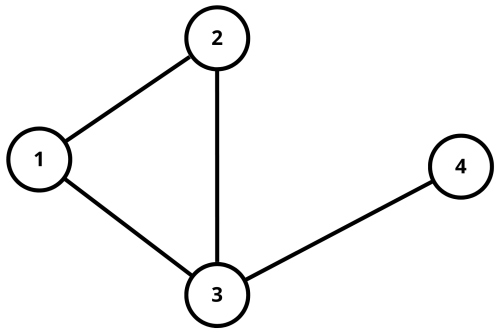


Fig. 4.7. Example for the Adjacent matrix

The Adjacent matrix is computed by giving a value to each connection between nodes. In the case of the graph of Figure 4.7, "1" is connected with "2" and "3" but not with "4", that means the first row of the matrix will be: $\begin{pmatrix} 0 & 1 & 1 & 0 \end{pmatrix}$. Following the same intuition, Adjacent matrix "A", in this case, will be:

$$A_{4x4} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{4.13}$$

Now that we know how to compute the Adjacency Matrix of a graph, let's take an easier example to start with the computations (See Fig. 4.8)

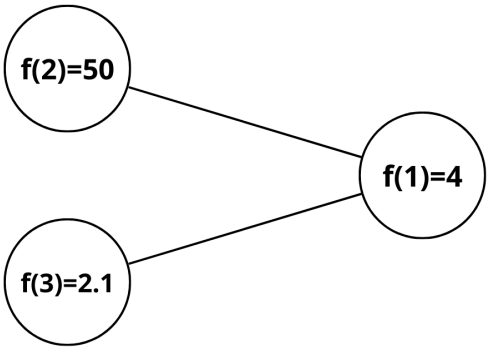


Fig. 4.8. Example we will use to follow the computations

We will start by computing the **Laplacian** as $A = I - D^{-1/2}AD^{-1/2}$ (see (4.7)).

- Degree matrix $D = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$; $D^{-1/2} = \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
- Adjacent matrix $A = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$
- The **Laplacian** matrix Δ will be:

$$\Delta = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & 1 & 0 \\ -\frac{1}{\sqrt{2}} & 0 & 1 \end{pmatrix} \tag{4.14}$$

In this way, it is easier to compute the **Eigenvalues** and **Eigenvectors** of the Laplacian:

$$det(\Delta - \lambda I) = 0 \rightarrow \begin{vmatrix} 1-\lambda & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & 1-\lambda & 0 \\ -\frac{1}{\sqrt{2}} & 0 & 1-\lambda \end{vmatrix} = 0; \lambda_1 = 2; \lambda_2 = 1; \lambda_3 = 0 \tag{4.15}$$

As we defined in (4.10), all eigenvalues lie between 0 and 2, i.e., $\lambda_i \in [0, 2],\ \forall i$. We can continue now obtaining the Eigenvectors as:

$$\Delta\phi_1^* = \lambda_1\phi_1^* \tag{4.16}$$
$$\Delta\phi_2^* = \lambda_2\phi_2^* \tag{4.17}$$
$$\Delta\phi_3^* = \lambda_3\phi_3^* \tag{4.18}$$

with $\phi_i^*,\ i = \{1, 2, 3\}$ being the first, second, and third eigenvector; respectively. In the following, we exemplify how to compute the first eigenvector ϕ_1^* for the Laplacian Δ defined in (4.14):

$$\begin{pmatrix} 1 & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & 1 & 0 \\ -\frac{1}{\sqrt{2}} & 0 & 1 \end{pmatrix} \begin{pmatrix} \phi_{1,1}^* \\ \phi_{1,2}^* \\ \phi_{1,3}^* \end{pmatrix} = 2 \begin{pmatrix} \phi_{1,1}^* \\ \phi_{1,2}^* \\ \phi_{1,3}^* \end{pmatrix} \tag{4.19}$$

If we want an orthonormal base, we need to normalize the eigenvectors $\phi_i = \frac{\phi_i^*}{\|\phi_i^*\|}$. Thus, the orthonormal basis would be $< \phi_1, \phi_2, \phi_3 >$.

$$\phi_1 = \frac{1}{2}\begin{pmatrix} -\sqrt{2} & 1 & 1 \end{pmatrix} \tag{4.20}$$

$$\phi_2 = \frac{1}{\sqrt{2}}\begin{pmatrix} 0 & -1 & 1 \end{pmatrix} \tag{4.21}$$

$$\phi_3 = \frac{1}{\sqrt{2}}\begin{pmatrix} \sqrt{2} & 1 & 1 \end{pmatrix} \tag{4.22}$$

With our orthonormal basis, given by the normalization of Eigenvectors in (4.19), we can get the Laplacian decomposed by them:

$$\Delta = \Phi^T \Lambda \Phi = \begin{bmatrix} -\frac{\sqrt{2}}{2} & \frac{1}{2} & \frac{1}{2} \\ 0 & -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ 1 & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}\begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} -\frac{\sqrt{2}}{2} & 0 & 1 \\ \frac{1}{2} & -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{1}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \tag{4.23}$$

Now, we have everything we need to perform the decomposition of f in **Fourier Series** (4.11).

$$f = \sum_{i=3}^{3} \langle \phi_i, f \rangle \phi_i = \langle \phi_1, f \rangle \phi_1 + \langle \phi_2, f \rangle \phi_2 + \langle \phi_3, f \rangle \phi_3 \tag{4.24}$$

Where $f = \begin{pmatrix} 4 & 50 & 2.1 \end{pmatrix}$ as we defined in Fig. 4.8.

$$= \langle \frac{1}{2}\begin{pmatrix} -\sqrt{2} & 1 & 1 \end{pmatrix}, \begin{pmatrix} 4 & 50 & 2.1 \end{pmatrix} \rangle \frac{1}{2}\begin{pmatrix} -\sqrt{2} & 1 & 1 \end{pmatrix} + ... \tag{4.25}$$

$$= \frac{1}{2}\left(-4\sqrt{2} + 50 + 2.1\right)\frac{1}{2}\begin{pmatrix} -\sqrt{2} & 1 & 1 \end{pmatrix} + ... \tag{4.26}$$

$$= \frac{52.1 - 4\sqrt{2}}{4}\begin{pmatrix} -\sqrt{2} & 1 & 1 \end{pmatrix} + \frac{2.1 - 50}{2}\begin{pmatrix} 0 & -1 & 1 \end{pmatrix} + \frac{4\sqrt{2} + 52.1}{2}\begin{pmatrix} \sqrt{2} & 1 & 1 \end{pmatrix} \tag{4.27}$$

Coming back to (4.11) we can make it more compact as:

$$f = \Phi\hat{f} = \begin{bmatrix} -\frac{\sqrt{2}}{2} & 0 & 1 \\ \frac{1}{2} & -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{1}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}\begin{bmatrix} \frac{52.1-4\sqrt{2}}{2} \\ \frac{2.1-50}{\sqrt{2}} \\ \frac{4\sqrt{2}+52.1}{\sqrt{2}} \end{bmatrix} = \mathscr{F}^{-1}(\hat{f}) \tag{4.28}$$

As $f = \Phi\hat{f} \rightarrow \Phi^T f = \Phi^T\Phi\hat{f} = \Phi^T f = \hat{f}$, the transposed of the normalized Eigenvector matrix of the **Laplacian**, multiplied by f, is the **Fourier Transform**.

If we want to do the convolution of a filter "w" over f, we can do it by the property we have just shown:

$$\mathscr{F}(w * f) = \mathscr{F}(w) \odot \mathscr{F}(f) \tag{4.29}$$

$$\mathscr{F}^{-1}(\mathscr{F}(w) * \mathscr{F}(f)) = \mathscr{F}^{-1}(\mathscr{F}(w) \odot \mathscr{F}(f)) \tag{4.30}$$

$$w * f = \Phi(\Phi^T w \odot \Phi^T f) \tag{4.31}$$

Previously, we have redefined the convolution for the case of graph by **Graph Spectral Theory**. There are many models that implement this type of spectral convolution.

VanillaGCN

Let's start from the simplest one. **Vanilla Spectral GCN** [39] is a raw implementation of the Spectral Convolution, with an activation function (like ReLU, hyperbolic tangent...) denoted with η as:

$$f^{l+1} = \eta(w^l * h^l) = \eta(\Phi(\Phi^T w^l \odot \Phi^T f^l)) \tag{4.32}$$

Where we can denote the superscript l as the l^{th} layer of the convolution. We will see in the implementation sections that we will carry out more than one convolution to improve performance.

This was the first **Spectral Convolution** technique to be implemented, but has many limitations:

- It has no guarantee of Spatial localization of filters.
- $O(n)$ parameters to learn. In this case, our algorithm follows a linear run time complexity[40] on the number of parameters to learn.
- $O(n^2)$ learning complexity, as it is a Fourier transform of a full matrix Φ.

This implementation is the core. From now on, what we are going to do is try to solve the previous limitations using different tools.

SplineGCN

In order to solve the issue of localized filters, with **Spline GCNs**[39][41] it is proposed to compute smooth special filters. This motivation comes from the *Heisenberg's uncertainty principle*: if you smooth in frequency domain, you get a smaller compact support.

For that reason, we will take an equation that will help us solve the issue of the smoothness exposed above: *Parseval's identity* (4.33).

$$\int_{-\infty}^{\infty} |x|^{2k} |w(x)|^2 \, dx = \int_{-\infty}^{\infty} |\frac{\partial^k \hat{w(\lambda)}}{\partial \lambda^k}| \, d\lambda \tag{4.33}$$

How can this equation helps us with the issue of smoothness? *Parseval's identity* states that smooth functions (i.e., small derivative on the right side) result in a smaller compact support [42] (e.g., on the left side $w(x) \simeq 0, x > 20$).

To achieve the state proposed above, in order to accomplish Spatial Location, we approximate the filter w using *Splines*. *Splines*[43] are polynomial functions defined piecewise that are differentiable and draw a curve in space. Thanks to that, we can achieve a small compact support according to *Parseval's Identity*(4.33), thus, a Spatial Location.

If we take $\hat{w}(\Lambda)$ from (4.12), we saw that it is a full diagonal matrix with n components as:

$$\hat{w}(\Lambda) = diag(\Phi^T w) = \begin{bmatrix} \hat{w}(\lambda_1) & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \hat{w}(\lambda_n) \end{bmatrix} \tag{4.34}$$

Therefore, vector $\hat{w}$ will be: $\hat{w} = \begin{pmatrix} \hat{w}(\lambda_1) \\ \vdots \\ \hat{w}(\lambda_n) \end{pmatrix}$

When we use *splines* (which will be denoted as "B"), we can simplify this computation because we will need to learn a significantly smaller number of parameters:

$$\hat{w}(\Lambda) = \sum_{i=0}^{k} B_i(\lambda_n) = \begin{pmatrix} B_0(\lambda_1) \\ \vdots \\ B_0(\lambda_n) \end{pmatrix} + \ldots + \begin{pmatrix} B_k(\lambda_1) \\ \vdots \\ B_k(\lambda_n) \end{pmatrix} \tag{4.35}$$

$$B_i(\lambda) = a_i + b_i(\lambda - x_i) + c_i(\lambda - x_i)^2 + d_i(\lambda - x_i)^3 \tag{4.36}$$

Where a_i, b_i, c_i, x_i are Splines parameters to be learned when approximating the filter $w(\lambda)$.

Let's say that for both cases we have $n = 10^6$:

- For the **VanillaGCN** case (4.31), we are going to have $n = 10^6$ parameters to learn.
- Now for the **SplineGCN** case (4.35) using $k = 100$ splines, we will have $5x100 = 500$ (5 scalars if we count a_i, b_i, c_i, d_i and all x_i, i.e., the spline scalar offset) which is significantly fewer parameters than in the first case.

LapGCN[44]

From the previous section, we have learned how to get **Spatial Location** using the *Parseval's identity* approximating w by utilizing Splines.

If we compute the spectral filter $\hat{w}(\Lambda)$ using Splines, we end up with the need to compute the convolution using eigenvectors: $w * f = \Phi\hat{w}(\Lambda)\Phi^T f$. This eigen-descomposition is computationally expensive. How can we avoid it? We will see that one simply way to do it is to learn the functions of the Laplacian directly from the graph.

To solve the eigen-descomposition issue, as mention previously, we can use the following intuition:

$$w * f = \hat{w}(\Delta)f = \sum_{k=0}^{k-1} w_k \Delta^k f = \sum_{k=0}^{k-1} w_k X_k \tag{4.37}$$

With $X_k = \Delta X_{k-1}$ and $X_0 = f$. Notice that $\Delta = I - D^{-1/2}AD^{-1/2}$ (4.7), thus, Δ does not require any eigen-descomposition.

Now that the issue of eigen-descomposition is solved, we can define Δw^k as a Laplacian heat operator, i.e., $\Delta w^1$is heating around a neighbourhood at distance k = 1, Δw^2 is heating around a neighborhood at distance k = 2, always with respect to a source node (In this case denoted with red See fig. 4.9)

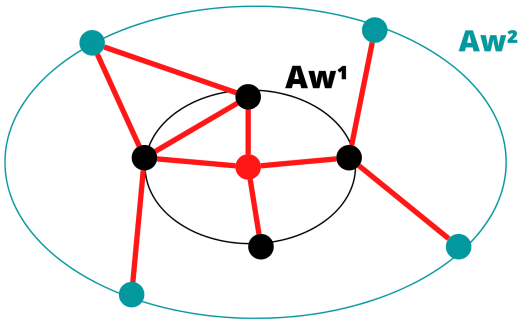


Fig. 4.9. Laplacian heat propagation around distance neighbours

In short, what we want to do is a small support, by learning our filter $w(\Delta)$ in neighborhoods at distance k, so filters will be localized in k-hops. In the example of a computer network, this means we will be able to predict a measure of traffic in the selected k-hops.

At this point, we have studied two ways to solve the issue of spatial localization for the filters. Now, we also know how can we lower the support of the function:

- **SplineGCN** (4.35): $\hat{w}(\Lambda) = \sum_{i=0}^{k-1} B_i \rightarrow \hat{w}\Delta f$ by making $\hat{w}$ smooth.
- **LapGCN** (4.37): $w * f = \hat{w}(\Delta)f = \sum_{k=0}^{k-1} w_k \Delta^k f$ by using a low k

ChebNets

After all LapGCN computations, we can observe one issue with Equation 4.37. We are approaching $\hat{w}(\Delta)$ with monomials Δ^k. Thus, basis of Δ^k are not orthogonal, i.e, will make impair in the learning stability.

Let's explain this with an example:

We have a pair of $\mathbb{R}^2$ orthogonal basis as $\{v_1 = (0,1), v_2 = (1,0)\}$ and another one but this time, non-orthogonal as $\{z_1 = (1,1), z_2 = (1,0)\}$. So $< (0,1),(1,0) >= 0$ and $< (1,1),(1,0) >= 1 \neq 0$. With this information, we will try to approximate the point $(2,1)$.

For the first case, with our orthogonal basis, $\{v_1 = (0,1), v_2 = (1,0)\}$, we can fit our filter as: $(2,1) = \sum_{i=1}^{2} w_i v_i = w_1(0,1) + w_2(1,0) = (w_2, w_1)$ (see Fig.4.10).

We can imagine this situation as if we had two levers (w_2, w_1), each of them will be pulled front or backward in order to catch the desired point.

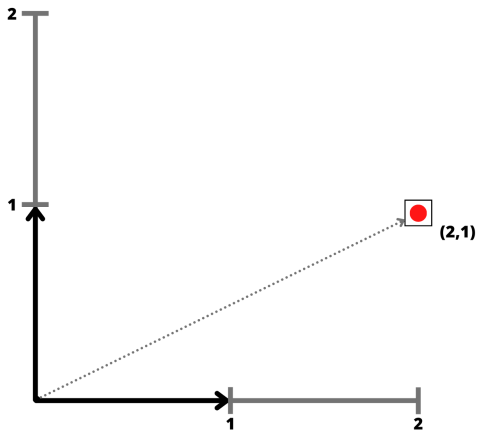


Fig. 4.10. Representation of approximation for the point (2,1) with orthogonal basis

This will be different for the second case: $(2,1) = \sum_{i=1}^{2} w_i v_i = w_1(1,1) + w_2(1,0) = (w_1 + w_2, w_1)$. So if we change one coefficient, this will change also the function approximation. Now we are not able to apply same lever intuition.

What this simple example means is that we need orthogonality if we want to learn with stability. At this point, in order to fulfill stability, we can use any orthonormal basis but has to have a *recursive equation* in order to speed up the calculus. Let's see why.

We will use **Chebyshev Polynomials**, as they are pretty common in signal processing. From (4.37):

$$w * f = \hat{w}(\Delta) f = \sum_{k=0}^{k-1} w_k \Delta^k f = \sum_{k=0}^{k-1} w_k T_k(\Delta) f \tag{4.38}$$

Where $T_k(\Delta)$ is the Chebyshev Polynomial. But how Chebyshev give us recursivity? We will start from (4.38)

$$w * f = \hat{w}(\Delta)f = \sum_{k=0}^{k-1} w_k T_k(\Delta)f = \sum_{k=0}^{k-1} w_k X_k \tag{4.39}$$

$$X_k = 2\tilde{\Delta}X_{k-1} \tag{4.40}$$

$$\tilde{\Delta} = 2\lambda_n^{-1}\Delta - I \tag{4.41}$$

As $X_0 = f, X_1 = \tilde{\Delta}f, \ldots, X_k = 2\tilde{\Delta}X_{k-1}$ and being λ_n the last eigenvalue.

At the end of the day, what we are doing is a multiplication of the Laplacian with one vector. We can now let the filter learn with stability due to the Chebyshev Polynomial.

From this point, let's try to prove that **VanillaGCN** are truncated **ChebNets**.

Keep (4.38) as starting point with an activation function η. We will take the following truncated values: $k = 2, w_0^l = w^l, w_1^l = -w^l, \lambda_2 = 2$

$$f^{l+1} = \eta(w^l * f^l) = \eta(\sum_{k=0}^{k-1} w_k T_k(\Delta)f^l) = \eta(w^l T_0(\Delta)f^l + (-w^l)T_1(\Delta)f^l) = \tag{4.42}$$

$$\eta(w^l(T_0(\Delta) - T_1(\Delta))f^l) = \eta(w^l(I + D^{-1/2}AD^{-1/2})f^l) = \tag{4.43}$$

$$\eta(w_{dx1}^l \hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}f^l) = \eta(\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}f^l W_{dxd}^l) \tag{4.44}$$

Notice that $\hat{A} = A + I$. So we will compute the mean for each node as:

$$f_i^{l+1} = \eta(\frac{1}{\hat{d}_i}\sum_{j \in N_i} \hat{A}_{ij} W^l f_j^l) \tag{4.45}$$

Expression that is exactly the same as the **Spatial Convolution** explanation for the **VanillaGCN** exposed above.

Study of accuracy with MNIST Dataset

Defferrard, Breson and Vandergherynst, at *Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering* [45], tried to implement all cases above with the goal to compare performance in all above GCN cases, using the MNIST Dataset [46]. They get into a limitation for ChebGCN, after a good performance (Accuracy : 99.14%). Chebnets are only supported by **Isotropic** models. Why? Because in **Spectral Convolutional Networks**, there is no notion of directions (as we use arbitrary graphs) [45]. Thus, the value of the filter will be the same in all directions for cycles with the same radius.

5. IMPLEMENTATION OF A GCN

5.1. Introduction to the project

In previous chapters, we have learned all the necessary context to deeply understand what is going on inside a GCN. Now we will try to bring this knowledge into the ground.

From now on, we will do is to lead readers into the creation of a GCN model. Our main goal is to get to understand it from the basis, so that we can develop a more complex model in the future. In order to that, we tried to simplify this model as much as possible.

We briefly recommend checking the *GitHub* repository, where all code is updated, before and after the reading of this thesis. Feel free to download the code and commit any improvement to the performance.

https://github.com/alejandrocalvillo/VanillaGCN

As you can see, this project has been developed in *Python*. With this programming language, it can be easier to work in Machine Learning models, specially with Neural Networks.

In addition to python, we used an assortment of libraries for different purposes that we will cite below:

- **PyTorch** [47]. *"An open source machine learning framework that accelerates the path from research prototyping to production deployment"*. This is the main library of the project.
- **PyTorch Geometric**. From the ecosystem of PyTorch, PyTorch Geometric is a library that helps with the implementation of irregular input data, as can be graphs.
- **Networkx** [48]. This library is used for the study of graphs with Python.
- **Matplotlib** [49]. All figures representing the MSE are plotted using this library.
- **NumPy** [50]. This library is mainly used to work with arrays and tensor, in order to adapt the data to fit into the model.

You can also find all these libraries and requirements attached in the *README.md* file of the project and also how to install it (in *Linux*).

The motivation of this code is to create a *Regression* model using all knowledge we have learned in above chapters. We are not looking to achieve good accuracy (as this is a

state-of-the-art paradigm nowadays), just to present the reader about this new technology and how to build a GCN from scratch.

Our goal was to predict the **Average Delay** of a Computer Network, using as input features for each node the **Total Packets Generated** and the **Jitter**. We want to use the same topology for each graph, but different values for the features (See Fig.5.1). Each graph is composed by 9 nodes and 30 directed edges. We will talk about the dataset in more detail in futures sections.

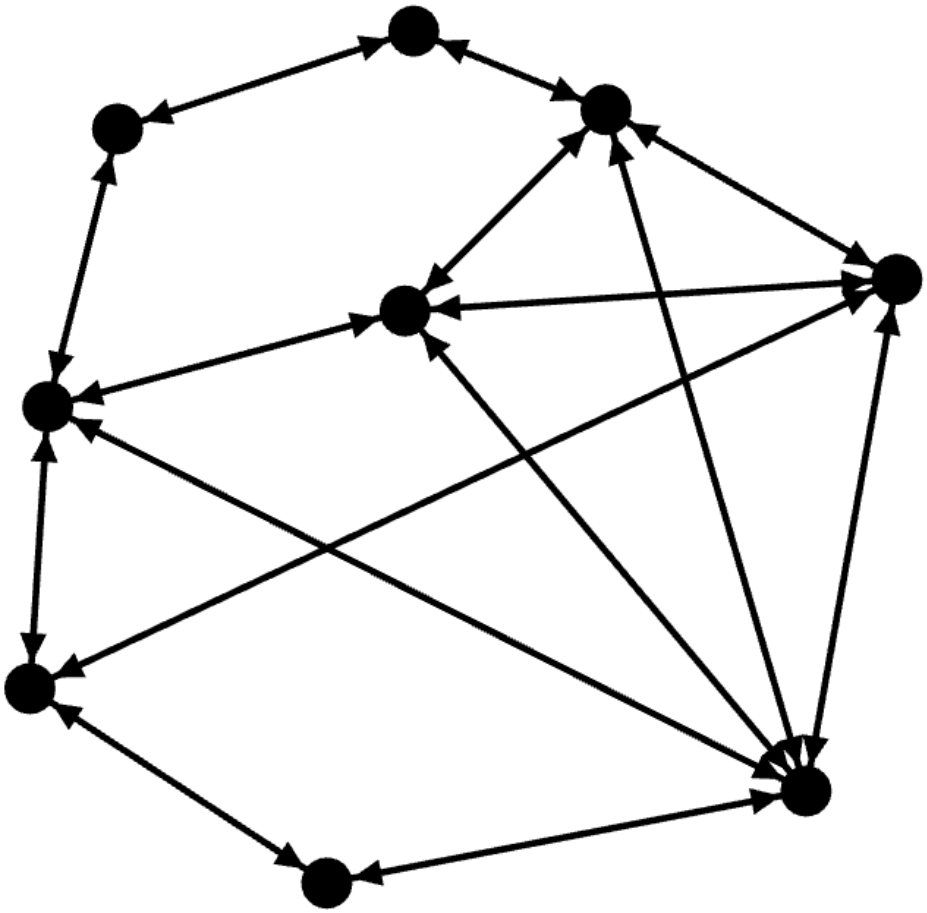


Fig. 5.1. Topology of our dataset graphs

5.2. About the model

In order to achieve our objetive, we chose to implement the simplest GCN convolution. As we saw, this is the case of a **VanillaGCN** [39].

In truth, the closest real-life implementation for this model is the **GCNConv** [51] approximation. We will dig into this now.

The following equation is the **VanillaGCN** core expression:

$$f^{l+1} = \eta(f^l * w^l) = \eta(I - D^{-1/2}AD^{-1/2}f^l w^l) \tag{5.1}$$

But for **GCNConv** they propose to use the following one:

$$f^{l+1} = \eta(f^l * w^l) = \eta(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}f^l w^l) \tag{5.2}$$

Where $\tilde{A} = A + I_n$. At first glance, we can wrongly say that are the same expression. But it is more complex than that.

What *Kipf et al.*[51] did is use *Chebyshev polynomial* to approximate the convolution (5.3), and then they used it again to approximate the *Laplacian* (with $k = 0, 1$) (5.4) as:

$$f * w \approx \sum_{k=0}^{k-1} w_k T_k(\Delta) f \approx \tag{5.3}$$

$$w_0 f - w_1 D^{-1/2}AD^{-1/2}f \tag{5.4}$$

From this point, in order to reduce overfitting, they use fewer parameters. So they take $w_0 = -w_1 = w$ to get:

$$w * f \approx w(I_n + D^{-1/2}AD^{-1/2})f \tag{5.5}$$

As they said, using this *renormalization trick* $I_n + D^{-1/2}AD^{-1/2} \rightarrow \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$, we get (5.2).

Let's see the implementation for our case in code 5.1 (you can find this code into *model.py* file in my *GitHub* repository). As you can see, this is an easy example of how to construct a Graph Convolutional Network.

1. We need to declare how many *Convolutional Layers* we want to use (Lines 5 and 6). In our case, there will be two.

2. Inside each declared *Convolutional Layer*, we need to specify the number of features (2), the number of hidden state (64) and the number of targets (1).

3. After all previous steps, we are ready to declare the *forward* function (Line 8) with two parameters:

   - **x**: as the input data.
   - **edge_index**: as the adjacency matrix.

4. Inside the *forward* function, we will add our two *Convolutional Layers*.(Lines 11 and 14). The output of this convolution will be fed into a *ReLu* function (in order to avoid negative values), and the output of the *ReLu* will be the input of a dropout function.

5. When we are done with all the previous steps, we will return the output, as it will be the target we want to predict.

CÓDIGO 5.1. Python code for the model of the GCN

```
1  class MyGCN(torch.nn.Module):
2      def __init__(self):
3          super().__init__()
4          #2 inputs, 64 as hidden state
5          self.conv1 = GCNConv(2, 64, add_self_loops = False)
6          self.conv2 = GCNConv(64, 1, add_self_loops = False)
7
8      def forward(self, x, edge_index):
9          x, edge_index = x, edge_index
10
11         x = self.conv1(x, edge_index)
12         x = F.relu(x)
13         x = F.dropout(x, training=self.training)
14         x = self.conv2(x, edge_index)
15         x = F.relu(x)
16         x = F.dropout(x, training=self.training)
17
18         return x
```

5.3. The dataset

We wanted to test our model with a real-life dataset in order to approximate this thesis to computer networks state-of-the-art issues. For that reason, we need a dataset, graph examples that behave as Digital Twins.

In order to obtain a dataset that simulates a real-world example of a Computer Network (what we consider as a Digital Twin), we enrolled ourselves into the *Universitat Politècnica de Catalunya Graph Neural Networking challenge 2022* [2]

5.3.1. About the challenge

In this challenge, participants are provided with two things:

- A GNN-model for network performance evaluation, named **RouteNet-Fermi**[52].

- A packet-level node simulator based on **OMNeT++**[53].

This GNN, was provided with thousands of graphs of up to 10 nodes. After the training, the model was able to predict per-path delay for networks with the same number of nodes with a Mean Relative Error smaller than 5%.

You can find all the code of the challenge in their *GitHub* repository at:

https://github.com/BNN-UPC/GNNetworkingChallenge/tree/2022_DataCentricAI

The goal of the challenge was to achieve the same, or lower accuracy using a dataset with not a thousand but a hundred of samples.

5.3.2. Obtaining the dataset

As mentioned before, to obtain the dataset for the challenge, it is needed to use OMNeT++. This tool is a framework for building networks simulations.

We were given some code on Python that uses this tool as a **Docker**[54] image. Simulation of real-world networks computationally is really expensive. For that reason, the challenge provided us with a scale-down environment, in order to generate graph samples faster.

For simplicity, the challenge gives us a partial code implementation to use this tool in a **Jupiter Notebook**[55] format. You can find this notebook at their repository (exposed above) in the file named *quickstart.ipynb*. This file comes with a short installation guide about how to generate data and run experiments using **RouteNet-Fermi**. From this file, we will only use the part where they explain about the data generator

We will not dig into how OMNeT++ works internally because that is not the object of this study. We will only use the tools that the Challenge provides to us as a *"Black box"* machine. You can see a simple schematic of this process at Fig.5.2

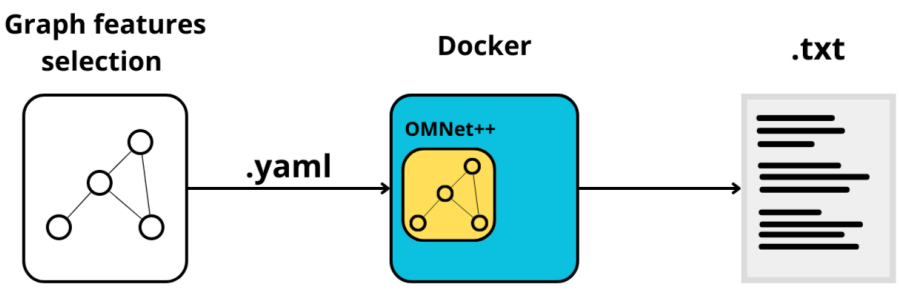


Fig. 5.2. Process of generating dataset using OMNet++

From now on, we will explain our code for the generation of the dataset (you can find this code into *generador_datos.py* file in my *GitHub* repository).

Firstly, we will need to declare the path where the dataset will be located, as well as the needed file (*simulation.txt*) and the name of the dataset. You can see the code at 5.2.

CÓDIGO 5.2. Select the path for storing the dataset

```
1  # Define destination for the generated samples
2  training_dataset_path = "Newtraining"
3  #paths relative to data folder
4  graphs_path = "graphs"
5  routings_path = "routings"
6  tm_path = "tm"
7  # Path to simulator file
8  simulation_file = os.path.join(training_dataset_path,"simulation.txt
       ")
9  # Name of the dataset: Allows you to store several datasets in the
       same path
10 dataset_name = "dataset1"
11
12 if (os.path.isdir(training_dataset_path)):
13     print ("Destination path already exists. Files within the
           directory may be overwritten.")
14 else:
15     os.makedirs(os.path.join(training_dataset_path,graphs_path))
16     os.mkdir(os.path.join(training_dataset_path,routings_path))
17     os.mkdir(os.path.join(training_dataset_path,tm_path))
```

Now, we can focus on generating a graph topology file, in order to process it in the following steps. As you can see in 5.3, we can choose about the QoS police and weights we want to use:

- *First In First Out* (FIFO): is a shared queue for all packets. The first packet to come is the first packet to be processed.
- *Strict Priory* (SP): we will have a queue for each *Type of Service* (ToS). Packets will be processed in priority order.
- *Weighted Fair Queueing* (WFQ): the goal is to get fairness in processing. We will have tree queues, with an assigned weight. In every iteration, the node will choose a queue to process, adding this weight plus the rate of the queue.
- *Deficit Round Robin* (DRR): same starting point as in WFQ, but DRR will cycle around the queues. The higher the weight, the more time it spends on this queue.

You can see (in line 16) we select FIFO policy. You can also select the buffer sizes of the nodes in bits. This value should be between 8000 and 64000 bits. We selected 3200 as you can observe (line 18).

We can also select the bandwidth of the links, as you may notice in line 38. The bandwidth value should be between 10 and 10000 bps.

CÓDIGO 5.3. Generate a topology for the graph

```
1  '''
2  Generate a graph topology file. The graphs generated have the
       following characteristics:
3  - All nodes have buffer sizes of 32000 bits and FIFO scheduling
4  - All links have bandwidths of 100000 bits per second
5  '''
6  def generate_topology(net_size, graph_file):
7      G = nx.Graph()
8      nodes = []
9      node_degree = []
10     for i in range(net_size):
11         node_degree.append(random.choices([2,3,4,5,6],weights
               =[0.34,0.35,0.2,0.1,0.01])[0])
12
13         nodes.append(i)
14         G.add_node(i)
15         # Assign to each node the scheduling Policy
16         G.nodes[i]["schedulingPolicy"] = "FIFO"
17         # Assign the buffer size of all the ports of the node
18         G.nodes[i]["bufferSizes"] = 32000
19
20     finish = False
21     while (True):
22         aux_nodes = list(nodes)
23         n0 = random.choice(aux_nodes)
24         aux_nodes.remove(n0)
25         # Remove adjacents nodes (only one link between two nodes)
26         for n1 in G[n0]:
27             if (n1 in aux_nodes):
28                 aux_nodes.remove(n1)
29         if (len(aux_nodes) == 0):
30             # No more links can be added to this node - can not
                   acomplish node_degree for this node
31             nodes.remove(n0)
32             if (len(nodes) == 1):
33                 break
34             continue
35         n1 = random.choice(aux_nodes)
36         G.add_edge(n0, n1)
37         # Assign the link capacity to the link
38         G[n0][n1]["bandwidth"] = 100000
39
40         for n in [n0,n1]:
41             node_degree[n] -= 1
42             if (node_degree[n] == 0):
43                 nodes.remove(n)
```

```
                if (len(nodes) == 1):
                    finish = True
                    break
        if (finish):
            break
    if (not nx.is_connected(G)):
        G = generate_topology(net_size, graph_file)
        return G

    nx.write_gml(G,graph_file)

    return (G)
```

Following the code, you may find a function that generates the shortest path routing for the topology we created before (code at 5.4). This function generates a *.txt* file where each line represents a sequence of nodes as a path.

CÓDIGO 5.4. Generate a topology for the graph

```
'''
Generate a file with the shortest path routing of the topology G
'''
def generate_routing(G, routing_file):
    with open(routing_file,"w") as r_fd:
        lPaths = nx.shortest_path(G)
        for src in G:
            for dst in G:
                if (src == dst):
                    continue
                path =  ','.join(str(x) for x in lPaths[src][dst])
                r_fd.write(path+"\n")
```

As a final step in order to generate all necessary files, we need to get a traffic matrix file. Inside this function, we can choose about:

- The source (*src*) and destination (*dst*) nodes to make the flow.
- The Average Bandwidth (*avg_bw*) for the previously chosen flow.
- *pkt_dist* is to specify the distribution that follows the flow to make a packet to appear (Poisson = 0, CBR = 1, ON-OFF = 2), *pkt_size_n* is the size of the packet (in bits), and *prob_n* is the relative probability with respect to the other sizes.
- The Type of Service (*tos*) designated to the packets generated for the selected flow.

You can find this at line 28 of the code in 5.5.

CÓDIGO 5.5. Generate a traffic matrix for the graph

```
'''
Generate a traffic matrix file. We consider flows between all nodes
    in the newtork, each with the following characterstics
- The average bandwidth ranges between 10 and max_avg_lbda
- We consider three time distributions (in case of the ON-OFF policy
     we have off periods of 10 and on periods of 5)
- We consider two packages distributions, chosen at random
- ToS is assigned randomly
'''
def generate_tm(G,max_avg_lbda, traffic_file):
    poisson = "0"
    cbr = "1"
    on_off = "2,10,5" #time_distribution, avg off_time exp, avg
        on_time exp
    time_dist = [poisson,cbr,on_off]

    pkt_dist_1 = "0,300,0.5,1700,0.5" #genric pkt size dist,
        pkt_size 1, prob 1, pkt_size 2, prob 2
    pkt_dist_2 = "0,500,0.6,1000,0.2,1400,0.2" #genric pkt size dist
        , pkt_size 1, prob 1,
                                               # pkt_size 2, prob 2,
                                                    pkt_size 3, prob
                                                    3
    pkt_size_dist = [pkt_dist_1, pkt_dist_2]
    tos_lst = [0,1,2]

    with open(traffic_file,"w") as tm_fd:
        for src in G:
            for dst in G:
                avg_bw = random.randint(10,max_avg_lbda)
                td = random.choice(time_dist)
                sd = random.choice(pkt_size_dist)
                tos = random.choice(tos_lst)

                traffic_line = "{},{},{},{},{},{}".format(
                    src,dst,avg_bw,td,sd,tos)
                tm_fd.write(traffic_line+"\n")
```

Finally, we will merge all the files we have generated in order to get the dataset (code at 5.6). There will be 100 samples with 5 different topologies and 20 traffic matrices, with the number of nodes between 6 and 10 (for our case we will only need the graphs with 9 nodes), and the maximum average bandwidth per flow (*max_avg_lbda*) up to a 10000.

After all this, we can call the *Docker* image to start working (From line 25 to 51) by generating the configuration file.

CÓDIGO 5.6. Generate the dataset with all generated files

```
"""
```

```
We generate the files using the previously defined functions. This
    code will produce 100 samples where:
- We generate 5 topologies, and then we generate 20 traffic matrices
    for each
- The topology sizes range from 6 to 10 nodes. Lo cambio a 9
- We consider the maximum average bandwidth per flow as 1000
"""
max_avg_lbda = 1000
with open (simulation_file,"w") as fd:
    for net_size in range (9,8): #Antes [6,11] ahora 9
        #Generate graph
        graph_file = os.path.join(graphs_path,"graph_{}.txt".format(
            net_size))
        G = generate_topology(net_size, os.path.join(
            training_dataset_path,graph_file))
        # Generate routing
        routing_file = os.path.join(routings_path,"routing_{}.txt".
            format(net_size))
        generate_routing(G, os.path.join(training_dataset_path,
            routing_file))
        # Generate TM:
        for i in range (20):
            tm_file = os.path.join(tm_path,"tm_{}_{}.txt".format(
                net_size,i))
            generate_tm(G,max_avg_lbda, os.path.join(
                training_dataset_path,tm_file))
            sim_line = "{},{},{}\n".format(graph_file,routing_file,
                tm_file)
            # If dataset has been generated in windows, convert
                paths into linux format
            fd.write(sim_line.replace("\\","/"))


# First we generate the configuration file
conf_file = os.path.join(training_dataset_path,"conf.yml")
conf_parameters = {
    "threads": 6,# Number of threads to use
    "dataset_name": dataset_name, # Name of the dataset. It is
        created in <training_dataset_path>/results/<name>
    "samples_per_file": 10, # Number of samples per compressed file
    "rm_prev_results": "n", # If 'y' is selected and the results
        folder already exists, the folder is removed.
}

with open(conf_file, 'w') as fd:
    yaml.dump(conf_parameters, fd)


def docker_cmd(training_dataset_path):
    raw_cmd = f"docker run --rm --mount type=bind,src={os.path.join(
        os.getcwd(),training_dataset_path)},dst=/data bnnupc/netsim:
```

```
        v0.1"
40      terminal_cmd = raw_cmd
41      if os.name != 'nt': # Unix, requires sudo
42          print("Superuser privileges are required to run docker.
                Introduce sudo password when prompted")
43          terminal_cmd = f"echo {getpass()} | sudo -S " + raw_cmd
44          raw_cmd = "sudo " + raw_cmd
45      return raw_cmd, terminal_cmd
46
47
48  # Start the docker
49  raw_cmd, terminal_cmd = docker_cmd(training_dataset_path)
50  print("The next cell will launch docker from the notebook.
        Alternatively, run the following command from a terminal:")
51  print(raw_cmd)
```

5.3.3. Adapting the data for a GCN

Before starting with the processing of the dataset in order to feed our model, we briefly recommend to look at the *preparation_dataset.py* file from my *GitHub*.

While to ready the data to be prepared, one main difficulty we found is that all features were stored in the links. As you may notice, we have been talking about flows between two nodes.

We needed to extract this information stored at the edges for it to became *"part of the nodes"* features, in order to fulfill the specifications of our model.

But let's start from the beginning (see code at 5.7). What we do first, is to select from all the dataset we have, applying two filters:

- *The Maximum Average Bandwidth Range* (max_avg_lamda_range) to be into the values from 10, to 10000 (Line 5).
- *The Number of Nodes of the Graph* (net_size_list) to be 9 in order to fullfill initial conditions (Line 7).

From here, we call a function given by the Challenge, named *DatanetAPI* (Line 9). This function will take, as input values, the source path (*src_path*) where the dataset is stored and the filters we chose above. The output of this function is a set of graphs that fulfill with the specifications made above.

What will happen is that the data from the edges will be extracted and stored in the nodes (From lines 24 to 32 and 34 to 37). How have we done this? Simply by adding all delay (*"AvgDelay"*), all Jitter (*"Jitter"*) and all Packets Generated (*"TotalPktsGen"*) for each flow with respect to a node. The node will be carrying this computed information.

Remember, our goal is to predict the Average Delay of each node using as input features the Total Packets Generated and the Jitter for each node.

With all this data we codify a tensor to store it, with the labels:

- **Total Packets Generated** (pkts_gen_lst) in position 0.
- **Jitter** (jitter_lst) in position 1.
- **Average Delay"** (delays_lst) in position 2.

We also stored a set of *Adjacency Matrices* (*edge_index*), so we can select the topology of the graph.

CÓDIGO 5.7. Prepare the data to be processed

```
def preparation_dataset(src_path):

    # Range of the maximum average lambda | traffic intensity used
    #  max_avg_lambda_range = [min_value,max_value]
    max_avg_lambda_range = [10,10000]
    # List of the network topology sizes to use
    net_size_lst = [9]
    # Obtain all the samples from the dataset
    reader = datanetAPI.DatanetAPI(src_path,max_avg_lambda_range,
        net_size_lst)
    samples_lst = []
    in_data = []
    out_data = []
    edge_index = []

    for sample in reader:
        samples_lst.append(sample)
        S = sample.get_performance_matrix()
        R = sample.get_traffic_matrix()

        input_to_tensor = []
        delays_lst = []
        jitter_lst = []
        pkts_gen_lst = []
        for i in range (sample.get_network_size()):
            cumulativeDelay = 0
            cumulativeJitter = 0
            cumulativePkts_gen = 0
            for j in range (sample.get_network_size()):
                if (i == j):
                    continue
                cumulativeDelay = S[i,j]["AggInfo"]["AvgDelay"] +
                    cumulativeDelay
                cumulativeJitter = S[i,j]["AggInfo"]["Jitter"] +
                    cumulativeJitter
```

```
33
34              for j in range (sample.get_network_size()):
35                  if (i == j):
36                      continue
37                  cumulativePkts_gen = R[i,j]["AggInfo"]["TotalPktsGen
                        "] + cumulativePkts_gen
38              #Node i
39              delays_lst.append(cumulativeDelay)
40              jitter_lst.append(cumulativeJitter)
41              pkts_gen_lst.append(cumulativePkts_gen)
42
43          #InputData Target delay
44          aux_in_lst = [pkts_gen_lst, jitter_lst, delays_lst]
45          metricas_in = np.asarray(aux_in_lst)
46          input_to_tensor = torch.Tensor(metricas_in)
47          in_data.append(input_to_tensor)
48
49          #Adjacency Matrix
50          G = nx.DiGraph(sample.get_topology_object())
51          edge_index.append(nx.adjacency_matrix(G))
52
53      in_data_tensor = torch.stack(in_data)
54
55      return in_data_tensor, edge_index
```

What remains to be done is to fit the data to the model. In order to do so, we developed a function exposed at the code from 5.8.

This function was made to condense information in the *torch_geometric.data* format, to create a *data* object (line 16). In this function, we accomplish the specification for the Adjacency Matrix to be inserted in the model (from line 12 to 14).

CÓDIGO 5.8. Model the data in order to fulfill specifications

```
1  def prepare_data(input, edge_index, labels):
2      """
3      Prepare data for the GCN model
4      Args:
5          input: Input metrics for the model (Tensor)
6          edge_index: All Adjacency Matrices in the requiered form for
               GCNConv (Tensor)
7          labels: Output data to compare (Tensor)
8      Returns:
9          data: The data ready to be fed to the GCN model
10     """
11     #For the case of a GCNConv, we will only use one topology
12     a = edge_index[0].todense()
13     edge_tensor = torch.tensor(a, dtype = torch.long)
14     input_edge_tensor = edge_tensor.nonzero().t().contiguous()
15
```

```
16      data = Data(x=input, edge_index=input_edge_tensor, y=labels)
17
18      return data
```

At this point, we got everything we need to train and test the model. This will be explained in the next section.

5.4. Training the model

First, we started by making some specifications of where the dataset was stored (code at 5.9). We loaded our function *preparation_dataset* with the source path (*src_path*) in order to make it return the metrics we specified in the previous section. This metrics were stored into *metricas_entrada* and the set of Adjacency Matrices into *edge_index*.

CÓDIGO 5.9. Path for the dataset

```
1  #Load data from BCN-GNN-CHALLENGE
2  data_folder_name = "training"
3  src_path = f"{data_folder_name}/results/dataset1/"
4  # data_folder_name = "checkpoint"
5  # CHECKPOINT_PATH = f"{data_folder_name}/checkpoint1"
6
7  #Load the data
8  metricas_entrada,edge_index = preparation_dataset(src_path)
```

What we needed to do now is to prepare the data to be processed (See code at 5.10), Firstly, we had to select the corresponding features (lines 7 and 8), using *input* as the input tensor with features *Total Packets Generated* and*Jitter*, and *label* for the target to compare, *Delay* feature. After this step, we needed to normalize the metrics. For that task, the *torch_funcional normalize function* (line 4).

We have 20 simulation graphs, such as $f_1, f_2..., f_{20}$ where the input of the graph in the i^{th} simulation is $f_i : G \mapsto \mathbb{R}^{9\times2}$, that is, we have as input something like:

$$f_i(G) = \begin{pmatrix} 10 & 21 \\ 10 & 21 \\ 8.5 & 12.2 \\ 7 & 1.4 \\ 1 & 21 \\ 0 & 21 \\ 12 & 21 \\ 8 & 21 \\ 10 & 1 \end{pmatrix} \tag{5.6}$$

For the target, there will be the same $g_1, g_2, ..., g_20$, but $g_i : G \mapsto \mathbb{R}^{9\times1}$.In order to

fulfill with the specification, we had to reshape the tensors from $(20, 2, 9)$ to $(20, 9, 2)$ in the case of the input data, and $(20, 1, 9)$ to $(20, 1, 2)$ in the case of the target one. Finally, all this data and the set of Adjacency Matrices were transferred as input to *prepare_data* function in order to create a *data* object (line 15).

The goal was to come up with a filter(s), w_i that inferred the *Delay* (g), with $\hat{g}$ being the output of the model as $\hat{g} = ReLU(w_2 * ReLU(w_1 * f))$, in order to become $\hat{g}$ as close as possible to g.

CÓDIGO 5.10. Normalization reshaping and creation of a data object

```
1  # Normalize data
2  # https://pytorch.org/docs/stable/generated/torch.nn.functional.
       normalize.html#torch-nn-functional-normalize
3
4  metricas_entrada = F.normalize(metricas_entrada)
5
6  #Select features to predict
7  input = metricas_entrada[:,:2,:] #train
8  labels =metricas_entrada[:,2,:] #train
9
10 #Reshape data in order to fulfill specified shape
11 input = np.reshape(input, (20, 9, 2))
12 labels = np.reshape(labels, (20, 9, 1))
13
14 #Prepare the dataset
15 data = prepare_data(input=input, edge_index=edge_index, labels=
       labels))
```

When you continue reading, you will come across with the declaration of a class named *SimpleCustomBatch* (code at 5.11). This is used in *DataLoader*, a function of the packet *torch.utils.data*, needed to create a batch of graphs. This class will helps us create a mini-batch of tensors.

CÓDIGO 5.11. Auxiliary class to form a mini-batch of tensors

```
1  class SimpleCustomBatch:
2      def __init__(self, data):
3          data = prepare_data(input=input, edge_index=edge_index,
               labels=labels)
4          self.inp = data.x
5          self.tgt = data.y
6
7      # custom memory pinning method on custom type
8      def pin_memory(self):
9          self.inp = self.inp.pin_memory()
10         self.tgt = self.tgt.pin_memory()
11         return self
12
13 def collate_wrapper(batch):
```

```
14      return SimpleCustomBatch(batch)
```

After this clarification, what we find next is the code exposed at 5.12. We loaded a tensor dataset to be processed using the function *TensorDataset* from the package *torch.utils.data*. This function, in turn, is loaded with the features input (*data.x*) and target (*data.y*) (line 1).

In order to get the best performance for the model, we create two arrays:

- The first will store a set of **Epoch** that the model will be trained with (line 4).
- The second will have a set of **Learning Rates** to test what is the best option for the model (line 5).

With each of these parameters, we will draw a plot to observe if the model is truly learning and which are the best parameters.

CÓDIGO 5.12. Tensor Dataset and a set of parameters to evaluate

```
1  dataset = torch.utils.data.TensorDataset(data.x, data.y)
2
3  #Select number of epoch and learning rate
4  epochs = [100, 500, 1000, 10000, 100000]
5  lrs = [0.1, 0.001, 1e-3, 1e-4, 1e-5, 1e-6, 1e-9]
6
7  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu'
       )
```

What comes next is the process of training (code at 5.13). For each epoch, we chose all the possible learning rates declared in the set (lines 1 and 2). With that learning rate, we declared an *Adam* optimizer.

Now it's time to explain the *Dataloader* (line 12). This function what mainly does is to get 4 random samples from the dataset. To do so, we need to pass as arguments, the *dataset*, the *batch_size = 4* in order to get 4 samples, and, to make it random, *shuffle = true*. The *collate_fn* it is needed in order to merge a set of samples that are *Tensors*.

When all of the above had been successfully declared, it was time to start with the real training. Focus on line 17 and 18. These pieces of code will compute the prediction (line 17) and store it in a variable called *out*. Then, this prediction will be compared using a *Mean Squared Error* (*mse_loss*) as:

$$MSE = \frac{1}{N}\sum_{i=i}^{N}(g_i - \hat{g}_i)^2 \tag{5.7}$$

Where N is the total number of samples, g is the real value and $\hat{g}$ is the predicted value.

Our model has been codified to learn from this error. To achieve so, we back-propagated (*loss.backward*) this loss up to the model, so that we could make our GCN learn. (line 21).

When an iteration of this process is finished, we save the weights of the model (in order to store them for future prediction tasks) and plot the results. This results will be discussed in the next section.

CÓDIGO 5.13. Process of training and back-propagation of the model

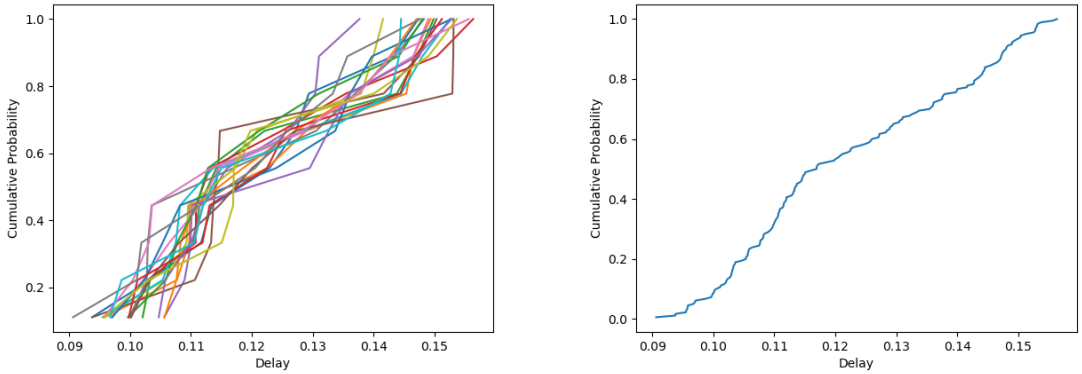
```
1  for epoch in epochs:
2      for lr in lrs:
3
4          model = MyGCN().to(device)
5          model.train(True)
6
7          optimizer = torch.optim.Adam(model.parameters(), lr=lr)
8          iterations = 0
9          loss_ar = []
10         for i in range(epoch):
11
12             testloader = torch.utils.data.DataLoader(dataset,
                   batch_size=4, collate_fn= collate_wrapper, shuffle=
                   True)
13
14             for j, data_in in enumerate(testloader):
15                 optimizer.zero_grad()
16                 iterations = iterations + 1
17                 out = model(x = data_in.inp , edge_index=data.
                       edge_index)
18                 loss = F.mse_loss(out, data_in.tgt)
19                 mse_loss = loss.detach().numpy()
20                 loss_ar.append(mse_loss)
21                 loss.backward()
22                 optimizer.step()
23
24         state_dict = model.state_dict()
25         torch.save(state_dict, 'weigths/model_weights'+str(epoch)+
               str(lr)+'.pt')
26         plot_mse_epoch(iterations, loss_ar, epoch, lr)
```

5.5. Results

With the plots we have obtained in the previous sections, we will now discuss the training. As we said before, we make a plot for each pair of parameters. You can take a look at all the plots in the folder *mse_loss_plots* from the repository of the project.

But first, allow us to consider at the *Cumulative Distribution Functions* (CDF) from

the Delay feature, the one we want to predict.



(a) CDF of the delay feature for each graph in seconds(b) CDF of the delay feature for all graphs in seconds

Fig. 5.3. Delay Cumulative Distribution Function representation

We can find, in Fig. 5.3a, the delay represented in different colors, one by each graph. This delay is measured in seconds. As we can see, values are around 150ms and 90ms (for each node). Remember, the goal of the model is to be trained to predict this feature

Out of all the plots (you can find 29 different plots at the folder mentioned before), we selected the one you can see at Fig. 5.4. This model has been loaded with a *Learning Rate* of $1e^{-}4$ and trained during 100 Epoch. This gives us 500 iterations due to, as explained above, the batch is being given 4 random graphs from the 20 graphs our dataset is composed of. For each *epoch*, the model is fed with these 4 graphs one by one, in order to perform the training.

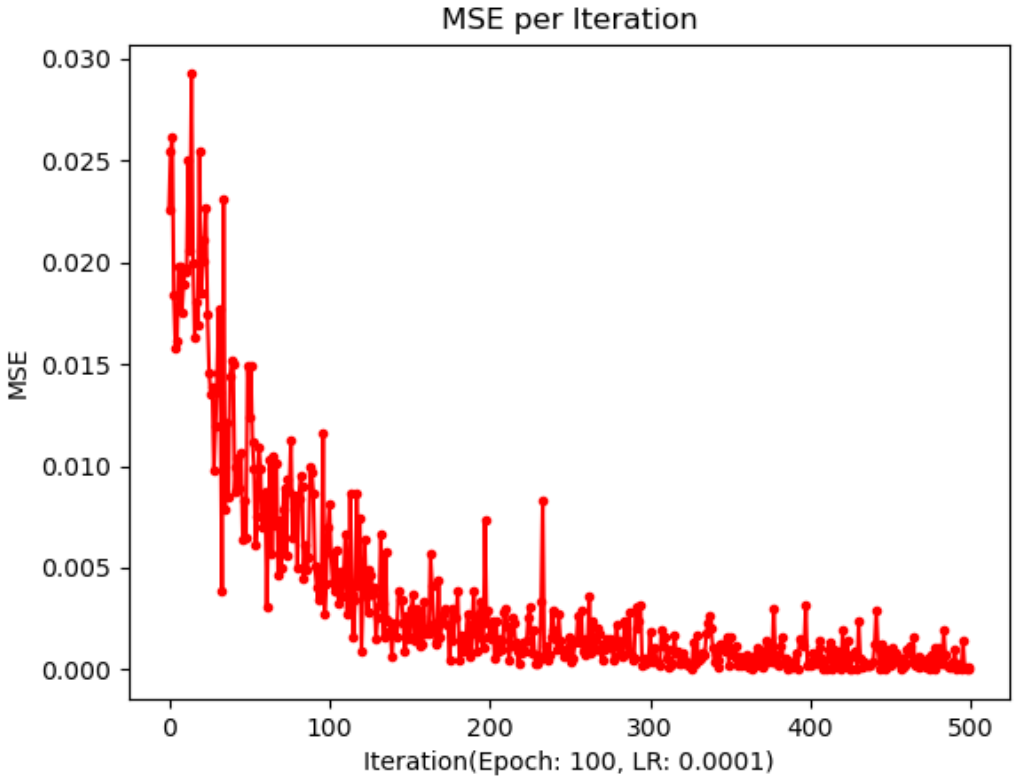
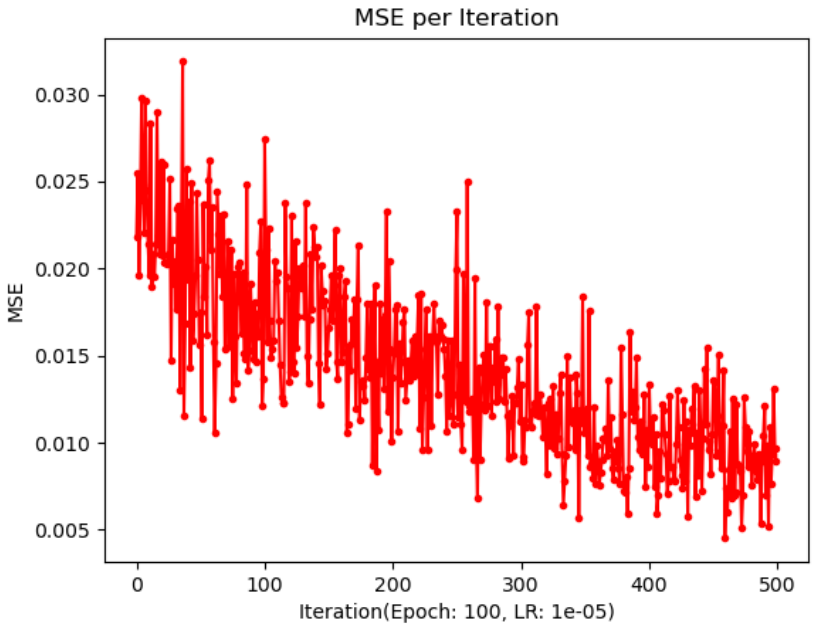


Fig. 5.4. Model trained with an Epoch of 100 and a Learning Rate of $1e^{-}4$ plot

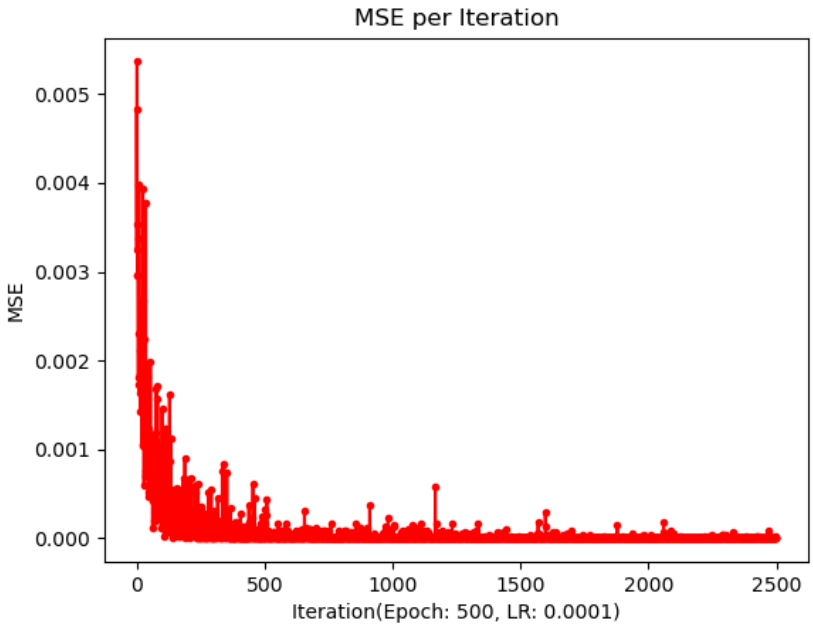
From Fig 5.4 we can determine that our model is successfully learning. To demonstrate that, we can take a look at the MSE axis. As iterations increase, the MSE is decreasing. When the model arrives to the 500th iteration, we stop the training in order to avoid overfitting.

If we want to know how far from the true value our model is predicting, we should see the MSE values at the point we want. For example, in the last iterations (> 400) we have a predicting error of $0.005s = 5ms$.

In order to discuss why we have selected these parameters for the model instead of others, we have chosen two more plots to show. You can find them at Fig.5.5



(a) Model trained with an Epoch of 100 and a Learning Rate of $1e^{-}5$ plot



(b) Model trained with an Epoch of 500 and a Learning Rate of $1e^{-}6$ plot

Fig. 5.5. Two discarded models with different parameters

The criteria we have chosen in order to discard models are mainly two:

- If the learning process of the model is too "scratchy", as if there was noise in this procedure.
- If the iterations are too long in order to get the MSE closer to 0, but not falling into

0 to drop into overfitting.

For that reasons, we have decided not only to discard Fig.5.5a and Fig.5.5b but also other combination of parameters.

By proving our model of a GCN is learning, **we have met the objective of this thesis**. Further steps (like testing or implementing the model in a service) are not the object of this study.

6. SOCIO-ECONOMIC ANALYSIS

We will divide this chapter in two different sections. **Estimation** cost of the project, where we will discuss financial costs; and **Social** analysis, where we will examine how this topic could help nowadays challenges.

6.1. Estimation

During this section, we will make an estimation of the real cost of developing this project. This estimation will be based in two key points: **Human Resources costs** and **Assets costs**. With this information, we determine that the cost of this project would be **8.255€**.

6.1.1. Human Resources cost

For the breakdown of human resources expenses, we have estimated the salaries of an internship student and an associate professor. The details can be seen in the following table:

Employee	**Salary**	**Time Spend (h)**	**Total**
Alejandro Calvillo	9€/h	620	5580€
Jorge Martin	15€/h	60	900€
			6480€

Table 6.1. HUMAN RESOURCES EXPENSES

As we can see the human resources cost would be **6480€**.

6.1.2. Assets cost

For the breakdown of assets expenses, we have estimated the price of a *Dell poweredge r740* server for running the simulations and a *laptop* to develop the code. Depreciation cost is estimated assuming all of them were bought on the 1th of September till 1th of February, when the project started.

Employee	**Cost**	**Depreciation**	**Total**
Dell poweredge r740	554,19€	6 months	500€
HP 15s-fq5066ns	750€	6 months	675€
			1175€

Table 6.2. ASSETS EXPENSES

With this information, we estimate that the asset costs would be **1175€**.

6.2. Social analysis

Over the last few days, we have been witness to the terrible outcomes of some natural disasters. Associated to this, there are always spikes in network traffic, leading to overloads that can potentially be harmful to the appropriate performance of the emergency services. This can result in a loss of lives and economic resources.

With a **Digital Twin**, where we previously had virtualized the network of the affected area, we can find and optimize solutions when we detect traffic congestions, helping to overcome difficulties. This tool, deployed with a **Graph Neural Network** model, is capable of identifying this type of situation, being able to orchestrate the network; for example, prioritizing emergency traffic services, such as 112.

7. PLANNING

In order to show the different tasks that compound this project, and their cost in matter of time, we have done a general *Gantt diagram* (See Fig. 7.1) to show the main duties. This thesis, started officially on the 1st of September, but before that date, the 1st of August, we already started looking for information on the subject.

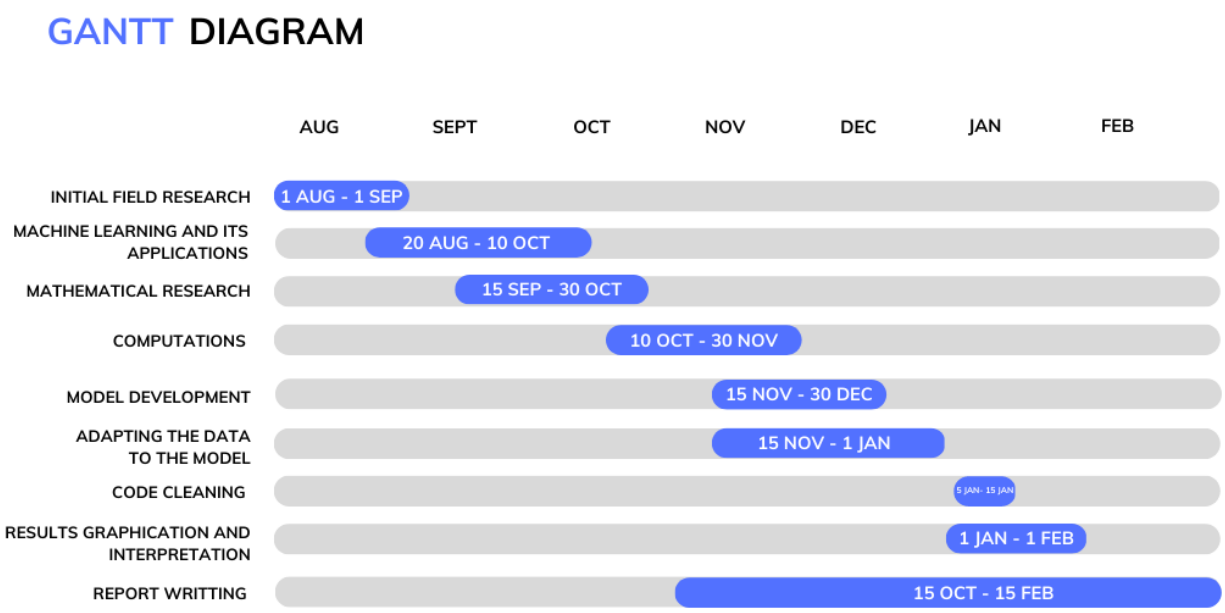


Fig. 7.1. Gantt diagram representing the different task in order to develop the project

This diagram does not show the number of hours of update meetings held. This time can be estimated by multiplying the *15 meetings* we have had by *2 hours* of duration $\approx 30h$.

The rest of the time spent on this thesis, could be estimated as: $3h$ a day per 30 days of work per 7 months $\approx 630h$.

8. CONCLUSION

Our main motivation for carrying through this study, was to take every reader of this thesis, closer to the world of Graph Neural Networks.

In order to do so, firstly, we explain some mathematical context in order to deeply understand what kind of computation is going on inside a GNN. When this context is successfully developed, we guide the reader towards the creation of a simple GCN machine learning model.

Building up this code, we had to deal with real challenges that machine learning developers have to face, such as the preparation of the dataset to be processed.

Now, we can conclude this journey though Graph Neural Networks. Although the goal of this study is not to evaluate the performance of the developed model, or to make a robuster one, futures investigations will find the solution of these two unresolved challenges.

This research, with the intention of giving the reader a closer point of view of how GNN could be some future in machine learning areas, may be terminated upon completion of the mentioned objectives. This project may be good a starting point for the reader to continue researching in this subject.

From this point, we can start thinking, not only about how GNN could be helpful for the coming technologies of telecommunications, as *Beyond-5G*[56] or *IEEE 802.11ax*[57] [58] new implementation are, but how we can bring the knowledge of how machine learning models work to every interested person.

BIBLIOGRAPHY

[1] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[2] J. Suárez-Varela *et al.*, "The graph neural networking challenge: A worldwide competition for education in ai/ml for networks," *ACM SIGCOMM Computer Communication Review*, vol. 51, no. 3, pp. 9–16, 2021.

[3] W. Jiang, B. Han, M. A. Habibi, and H. D. Schotten, "The road towards 6g: A comprehensive survey," *IEEE Open Journal of the Communications Society*, vol. 2, pp. 334–366, 2021.

[4] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.

[5] A. A. Soofi and A. Awan, "Classification techniques in machine learning: Applications and issues," *J. Basic Appl. Sci*, vol. 13, pp. 459–465, 2017.

[6] D. Maulud and A. M. Abdulazeez, "A review on linear regression comprehensive in machine learning," *Journal of Applied Science and Technology Trends*, vol. 1, no. 4, pp. 140–147, 2020.

[7] T. G. Dietterich, P. Domingos, L. Getoor, S. Muggleton, and P. Tadepalli, "Structured machine learning: The next ten years," *Machine Learning*, vol. 73, pp. 3–23, 2008.

[8] Y. Zhao *et al.*, "A survey of networking applications applying the software defined networking concept based on machine learning," *IEEE Access*, vol. 7, pp. 95 397–95 417, 2019.

[9] P. Cunningham, M. Cord, and S. J. Delany, "Supervised learning," *Machine learning techniques for multimedia: case studies on organization and retrieval*, pp. 21–49, 2008.

[10] J. Jeong, S. Lee, J. Kim, and N. Kwak, "Consistency-based semi-supervised learning for object detection," *Advances in neural information processing systems*, vol. 32, 2019.

[11] I. Giotis *et al.*, "Med-node: A computer-assisted melanoma diagnosis system using non-dermoscopic images," *Expert systems with applications*, vol. 42, no. 19, pp. 6578–6585, 2015.

[12] M. Vijh, D. Chandola, V. A. Tikkiwal, and A. Kumar, "Stock closing price prediction using machine learning techniques," *Procedia computer science*, vol. 167, pp. 599–606, 2020.

[13] C. Stanfill and D. Waltz, "Toward memory-based reasoning," *Communications of the ACM*, vol. 29, no. 12, pp. 1213–1228, 1986.

[14] V. Y. Kulkarni and P. K. Sinha, "Pruning of random forest classifiers: A survey and future directions," in *2012 International Conference on Data Science & Engineering (ICDSE)*, IEEE, 2012, pp. 64–68.

[15] W. S. Noble, "What is a support vector machine?" *Nature biotechnology*, vol. 24, no. 12, pp. 1565–1567, 2006.

[16] P. Dayan, M. Sahani, and G. Deback, "Unsupervised learning," *The MIT encyclopedia of the cognitive sciences*, pp. 857–859, 1999.

[17] S. Bradtke and M. Duff, "Reinforcement learning methods for continuous-time markov decision problems," *Advances in neural information processing systems*, vol. 7, 1994.

[18] B. Kiumarsi, K. G. Vamvoudakis, H. Modares, and F. L. Lewis, "Optimal and autonomous control using reinforcement learning: A survey," *IEEE transactions on neural networks and learning systems*, vol. 29, no. 6, pp. 2042–2062, 2017.

[19] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[20] Z. Wu *et al.*, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[21] E. Markowitz *et al.*, "Graph traversal with tensor functionals: A meta-algorithm for scalable learning," *arXiv preprint arXiv:2102.04350*, 2021.

[22] S. S. Du *et al.*, "Graph neural tangent kernel: Fusing graph neural networks with graph kernels," *Advances in neural information processing systems*, vol. 32, 2019.

[23] K. Xu *et al.*, "Representation learning on graphs with jumping knowledge networks," in *International conference on machine learning*, PMLR, 2018, pp. 5453–5462.

[24] P. Veličković, R. Ying, M. Padovano, R. Hadsell, and C. Blundell, "Neural execution of graph algorithms," *arXiv preprint arXiv:1910.10593*, 2019.

[25] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko, "A gentle introduction to graph neural networks," *Distill*, 2021, https://distill.pub/2021/gnn-intro. DOI: 10.23915/distill.00033.

[26] C. Grima Ruiz, *En busca del grafo perdido: Matemáticas con puntos y rayas*, ser. Ariel. Editorial Ariel, 2021. [Online]. Available: https://books.google.es/books?id=EGRDEAAAQBAJ.

[27] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," in *Proceedings of 2010 IEEE international symposium on circuits and systems*, IEEE, 2010, pp. 253–256.

[28] J. Lerma, *Cómo se comunican las neuronas*, Apr. 2022. [Online]. Available: https://www.csic.es/es/ciencia-y-sociedad/libros-de-divulgacion/coleccion-que-sabemos-de/como-se-comunican-las-neuronas.

[29] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning," *The annals of statistics*, vol. 36, no. 3, pp. 1171–1220, 2008.

[30] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[31] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[32] S. Sukhbaatar, R. Fergus, *et al.*, "Learning multiagent communication with backpropagation," *Advances in neural information processing systems*, vol. 29, 2016.

[33] F. Monti *et al.*, "Geometric deep learning on graphs and manifolds using mixture model cnns," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 5115–5124.

[34] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the royal statistical society: series B (methodological)*, vol. 39, no. 1, pp. 1–22, 1977.

[35] P. Veličković *et al.*, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[36] X. Bresson and T. Laurent, "Residual gated graph convnets," *arXiv preprint arXiv:1711.07553*, 2017.

[37] F. R. Chung, *Spectral graph theory*. American Mathematical Soc., 1997, vol. 92.

[38] D. K. Hammond, P. Vandergheynst, and R. Gribonval, "Wavelets on graphs via spectral graph theory," *Applied and Computational Harmonic Analysis*, vol. 30, no. 2, pp. 129–150, 2011.

[39] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.

[40] J. A. Mañas, *Análisis de algoritmos: Complejidad*, 1997.

[41] M. Henaff, J. Bruna, and Y. LeCun, "Deep convolutional networks on graph-structured data," *arXiv preprint arXiv:1506.05163*, 2015.

[42] O. Knill, *Lecture 31*, https://people.math.harvard.edu/~knill/teaching/math22b2019/handouts/lecture31.pdf, 2019.

[43] L. Schumaker, *Spline functions: basic theory*. Cambridge University Press, 2007.

[44] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *Advances in neural information processing systems*, vol. 29, 2016.

[45] ——, "Convolutional neural networks on graphs with fast localized spectral filtering," *Advances in neural information processing systems*, vol. 29, 2016.

[46] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[47] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[48] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[49] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[50] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[51] T. N. Kipf and M. Welling, *Semi-supervised classification with graph convolutional networks*, 2016. DOI: 10.48550/ARXIV.1609.02907. [Online]. Available: https://arxiv.org/abs/1609.02907.

[52] M. Ferriol-Galmés *et al.*, "Routenet-fermi: Network modeling with graph neural networks," *arXiv preprint arXiv:2212.12070*, 2022.

[53] A. Varga, "Omnet++," *Modeling and tools for network simulation*, pp. 35–59, 2010.

[54] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[55] T. Kluyver *et al.*, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds., IOS Press, 2016, pp. 87–90.

[56] J. Kaur, M. A. Khan, M. Iftikhar, M. Imran, and Q. E. U. Haq, "Machine learning techniques for 5g and beyond," *IEEE Access*, vol. 9, pp. 23 472–23 488, 2021.

[57] Y. Zhang *et al.*, "Deepwiphy: Deep learning-based receiver design and dataset for ieee 802.11 ax systems," *IEEE Transactions on Wireless Communications*, vol. 20, no. 3, pp. 1596–1611, 2020.

[58] W. Wydmański and S. Szott, "Contention window optimization in ieee 802.11 ax networks with deep reinforcement learning," in *2021 IEEE Wireless Communications and Networking Conference (WCNC)*, IEEE, 2021, pp. 1–6.