POLITECNICO DI TORINO

Master's Degree in Communications and Computer Networks Engineering

Master's Degree Thesis

Machine Learning-Based Routing and Wavelength Assignment in Optical Networks

Supervisors

Prof. Andrea BIANCO

Prof. Cristina ROTTONDI

Candidate

Alessandro DE VITIS

April, 2021

Abstract

Recently, the wide area of Machine Learning (ML) has attracted the attention of the networking community to address different optical networking problems. Among these, the Routing and Wavelength Assignment (RWA) problem represents a fundamental task for handling connection requests on-demand in real-time within a dynamic network scenario. Given such problem belonging to the NP-complete class, and hence being not optimally solvable in the case of large scenarios, this thesis aims at applying innovative ML methods to address the RWA problem in acceptable cmputational times.
In particular, Graph Neural Network (GNNs) are specifically tailored to solve problems with a graph-based structure, and hence offer interesting alternatives to face the RWA problem according to a new paradigm. We rely on a particular GNN architecture, which is the Message-Passing Neural Network (MPNN), able to exploit the data graph structure by following the message-passing principle. When trained according to a supervised approach, i.e. when fed with labeled RWA examples, results show that the MPNN is able to achieve excellent performances within small and medium-large scenarios, with peak accuracies above 90%.

KEYWORDS: Machine Learning, Graph Neural Network, Routing and Wavelength Assignment, Optical Networks

Acknowledgements

This thesis represents for me the end of a long journey lasted almost six years, during which I have learned a lot and, most, I have changed a lot. Therefore, I would like to say a huge "thank you" to Politecnico di Torino for the extraordinary teachings and opportunities it has offered me during my university career. In addition, I definitely would like to say "thank you" to Prof.Bianco and Prof.Rottondi for having assigned me this project, for having believed in me and always supported through the thesis development.
Last but not least, from the deep of my heart I say "thank you": to my parents, which have allowed me to study without thinking at anything else; to all my friends forever, because they have always supported my choices during these years and have never run away; to all my class-mates, who have shared with me the happiest and the most difficult moments; to the city of Poznań and all the people I met there, because they have made my Erasmus experience unforgettable.

Table of Contents

List of Tables

List of Figures

Acronyms

WDM
Wavelength Division Multiplexing

WR
Wavelength-Routed

RWA
Routing and Wavelength Assignment

ILP
Integer Linear Programming

ML
Machine Learning

AI
Artificial Intelligence

SDN
Software Defined Networking

SLE
Static Lightpath Establishment

DLE
Dynamic Lightpath Establishment

R
Routing

WA

Wavelength Assignment

FF

First-Fit

RF

Random-Fit

LU

Least-Used

MU

Most-Used

SL

Supervised Learning

UL

Unsupervised Learning

RL

Reinforcement Learning

NN

Neural Network

ANN

Artificial Neural Network

CNN

Convolutional Neural Network

RNN

Recurrent Neural Network

GNN

Graph Neural Network

MLP

Multilayer Perceptrons

SVM

Support Vector Machines

GN Block

Graph Network Block

MPNN

Message-Passing Neural Network

NLNN

Non-Local Neural Network

SCE

Softmax Cross-Entropy

HPC

High Performance Computing

Chapter 1

Introduction

1.1 Background of the Project

Nowadays, worldwide society is largely based on the Internet. This is arguably the most important and revolutionary engineering product in the human history, and the contemporary world relies on the Internet for the great part of human activities.
In such a more and more Internet-based world scenario, it is evident that the increase of bandwidth demands of network users has become a critical challenge. To handle the problem, among all the possible multiplexing techniques, Wavelength Division Multiplexing (WDM) has been rapidly gaining acceptance as solution means in optical fiber networks.
The implementation of such a multiplexing technique leads to what today is referred to as Wavelength-Routed (WR) optical WDM networks. In a WR network, wavelengths are the key resource to route and switch information in the optical domain. In particular, end users communicate with one another via all-optical WDM channels, which are referred to as lightpaths.
In setting up a ligthpath between two users, a route must be selected and a wavelength must be allocated to the lightpath. The problem of finding routes and assigning wavelengths in a multi-user communication scenario in a WR network is referred to as Routing and Wavelength Assignment (RWA) problem. Depending on the network implementation, the problem might have different constraints to satisfy, but the one to be always respected is that two lightpaths that span one link in common cannot be allocated the same wavelength.
Given the objective of minimizing the usage of resources, or, in other words, to maximize the users access to the network, the RWA problem has revealed from the beginning not easy to solve due to its NP-completeness nature.

1.2 Motivation and Goals of the Project

The above explained RWA problem, according to the operational research literature and in particular in [1], has been proved to belong to the NP-complete class; this means it can be optimally solved by exact Integer Linear Programming (ILP) formulas, which, however, suffer from high computational complexity, requiring a huge amount of time to solve medium-size and large-size network topologies. To bypass the computational time problem, a large number of heuristic algorithms have also been proposed in the literature, offering faster solutions than ILP but rather sub-optimal.
Given these assumptions, the objective of this project is to explore a recently developed area to solve computational challenges, which is Machine Learning (ML), in order to solve the RWA problem in reasonable times and with enough acceptable results.
Indeed, the interest in Artificial Intelligence (AI) and, more specifically, in the area of ML has been increasing rapidly in the networking community in recent years. Examples of ML implementations can be found in the currently being developed Software Defined Networking (SDN) principle, on which 5G and future networks will rely.
Therefore, the project has been launched in the belief that ML tools can turn useful even in an important aspect of modern RW optical networks, which is the RWA problem. In particular, a specific type of ML model, called Graph Neural Network (GNN), is tested to understand whether this innovative area can offer an alternative way to bypass RWA NP-hardness.

1.3 Thesis Outline

The content of the thesis is organized in the following way.
Chapter 2 provides a detailed background on the RWA problem, discussing its formal definition and the proposed algorithms from literature.
Chapter 3 introduces ML approaches and models, with a particular highlight on GNNs and how these work.
Chapter 4 presents in detail our GNN framework, used during the thesis to apply ML techniques to the RWA problem.
Chapter 5 reports performance and time complexity numerical results in different networking scenarios, showing ML ability in solving the RWA problem and how this varies depending on the choice of some input parameters.
Chapter 6 presents the project final considerations and possible future works.

Chapter 2

Background on Routing and Wavelength Assignment

2.1 Introduction to the RWA Problem

In a WR network scenario, end users communicate with one another via all-optical WDM channels, which are referred to as lightpaths (Fig. 2.1).

A lightpath is therefore a virtual circuit which exploits optical technology, and in general spans one or more fiber links. Every lightpath is allocated a specific wavelength on any link which is part of its route; in the absence of wavelength converters, a lightpath must occupy the same wavelength all along its route. Such a property is referred to as wavelength-continuity constraint, and will be held valid through all the thesis. In Fig. 2.1, for example, one can observe one lightpath which has been allocated the wavelength λ_1, and another lightpath which has been allocated the wavelength λ_2.
Lightpaths are clearly an efficient end-to-end way to allow communication between nodes, but suffer a fundamental technological constraint: two lightpaths which span the same link cannot be allocated the same wavelength. Therefore, considering wavelengths as precious network resources, routing lightpaths and assigning them wavelengths in order to minimize the resources usage becomes a challenging task. In particular, given a WR network and a set of connections to establish, as [2] states, "the problem of setting up lightpaths by routing and assigning a wavelength to each connection is called the Routing and Wavelength Assignment (RWA) problem".
Based on the nature of the connection requests, the RWA problem might assume different forms. As one can read in [3], connection requests may be of three types: static, incremental and dynamic.

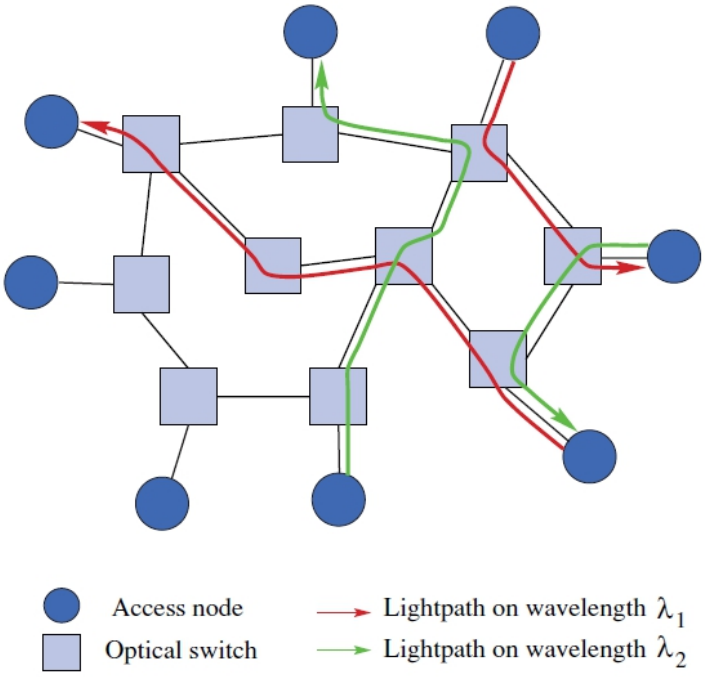


Figure 2.1: A WR network with lightpath connection [2].

- With static traffic, the entire set of connections is known in advance (often by means of a Traffic Matrix), and the RWA problem consists in setting up lightpaths for these connections in a global fashion while minimizing network resources such as the number of used wavelengths. Alternatively, one may attempt to set up as many of these connections as possible for a given fixed number of wavelengths.
  The RWA problem for static traffic is known as the Static Lightpath Establishment (SLE) problem, and will be analyzed in the next sections.

- In the incremental-traffic case, connection requests arrive sequentially, a lightpath is established for each connection, and the lightpath remains in the network indefinitely.
  This case is not of our interest and will not be considered.

- For the case of dynamic traffic, a time window is considered in which a sequence of connection requests arrives. If the connection (also known as call) request is accepted, then a lightpath is allocated (and a wavelength is reserved on the route) for all the connection duration, and is released, i.e. the wavelength is made again available, at the end of it.
  Given a set of available wavelengths, the objective is to set up lightpaths and

assign wavelengths in a manner that minimizes the amount of connection blocking. Such a problem is referred to as the Dynamic Lightpath Establishment (DLE) problem, and will be discussed in detail in the next sections.

The SLE problem can be formulated as a integer linear programming (ILP) problem, which is NP-complete [3].
For what concerns the DLE case, instead, the problem is definitely more difficult to solve due to the fact that connections are not known a-priori. Therefore, heuristics methods are generally employed and every incoming call is treated as a RWA problem to solve. As one can see deeper in the next sections, the heuristics methods tend to partition the RWA problem in two independent subproblems: Routing and Wavelength Assignment. Both the subproblems can be solved with several different heuristics algorithms from literature [2].

2.2 Static RWA Problem

In the static RWA problem, also known as SLE, all the connection requests are known in advance, and hence the problem can be solved in a global fashion off-line. Given a physical topology and a set of lightpaths to be established, the objective is to minimize the number of wavelengths needed to set up all the connections.
SLE, with the wavelength-continuity constraint (which is always kept valid), can be formulated as an "ILP in which the objective function is to minimize the flow in each link" [2]. Such a objective corresponds to minimizing the number of lightpaths passing through a particular link, which, in turn, leads to minimizing the number of needed wavelengths.
Let us define the following variables:

- λ_{sdw}: traffic (number of connection requests) from any source s to any destination d on any wavelength w; since we assume that, if two or more lightpaths exist between the same source-destination pair, each of them must employ a distinct wavelength, then it holds $\lambda_{sdw} \leq 1$.
- F_{ij}^{sdw}: traffic (number of connection requests) from source s to destination d on link ij and wavelength w; since a wavelength on a link can be assigned to only one path, it holds $F_{ij}^{sdw} \leq 1$.
- Λ: traffic matrix.
- Λ_{sd}: number of connections needed between source s and destination d.

Given a network physical topology and a set of wavelengths, the SLE problem can be formulated by means of the following ILP problem:

Minimize: F_{max}

such that:

- $F_{max} \geq \sum_{s,d,w} F_{max}^{sdw} \qquad \forall ij$

- $\sum_i F_{ij}^{sdw} - \sum_k F_{jk}^{sdw} = \begin{cases} \text{-}\lambda_{sdw} & \text{if } s = j \\ \lambda_{sdw} & \text{if } d = j \\ 0 & \text{otherwise} \end{cases}$ (2.1)

- $\sum_w \lambda_{sdw} = \Lambda_{sd}$

- $F_{ij}^{sdw} = 0{,}1$

- $\sum_{s,d} F_{ij}^{sdw} \leq 1$

For a given number of wavelengths, one can apply the ILP to see if a solution can be found. If a solution is not found, then a greater number of wavelengths is attempted. This procedure is iterated until the minimum number of wavelengths is found.
As already said in the previous section, the ILP formulation of the SLE problem has been proved being NP-complete [4], and hence becomes computationally consuming in case of medium-large size networks.
An interesting application of ML techniques to the SLE problem solved by the ILP formulation can be found in [5].

2.3 Dynamic RWA Problem

2.3.1 DLE Scenario

The DLE problem presents a scenario which is unarguably more realistic than the SLE one. In fact, as it is observed in reality, the connection requests are not known in advance, but rather it is considered a time window in which, at any time, a new call request can arrive, and, at any time, an active call can terminate.
Based on the network occupation and the adopted algorithm, a call request could either be accepted or be blocked. In the former case, a lightpath is set up for all the call duration and released at the end of it. Therefore, in that time interval, a specific wavelength is reserved for the call along its route. In case of blocking, instead, no resource is allocated and the call is dropped.
As for SLE, it is assumed that two lightpaths sharing at least one link cannot be allocated the same wavelength. The difference is that for the dynamic problem, instead of attempting to minimize the number of wavelengths as in the static case, we assume that the number of wavelengths is fixed, and we attempt to minimize the number of blocked calls.
Given that dynamic RWA is more complex than static RWA, it must be the case that dynamic RWA is also NP-complete [6]. Moreover, since it is not possible to know in advance when and which call requests will occur, DLE cannot be formulated as an ILP problem, but is rather decomposed in two subproblems: Routing (R) and Wavelength Assignment (WA).
R and WA subproblems can be solved in sequence for any incoming call request. The R subproblem outputs a route, which is given as input to the WA subproblem, whose output determines whether the call has been either accepted (and on which wavelength) or blocked.

2.3.2 R Subproblem

When a new connection request arrives, a route from source to destination must be determined first. In the literature, one can find various approaches to routing connection requests.

Fixed Routing

The most straightforward approach to routing a connection is to always choose the same fixed route for a given source-destination pair. The most popular choice is typically the shortest-path route, which can be computed by well-known algorithms such as Dijkstra's algorithm or Bellman-Ford algorithm.
The great advantage is that, given the network topology, a fixed route for any source-destination pair can be computed off-line and kept stored in the nodes

routing tables. Therefore, any connection request from source will be routed on the pre-determined route from source to destination (Fig. 2.2).

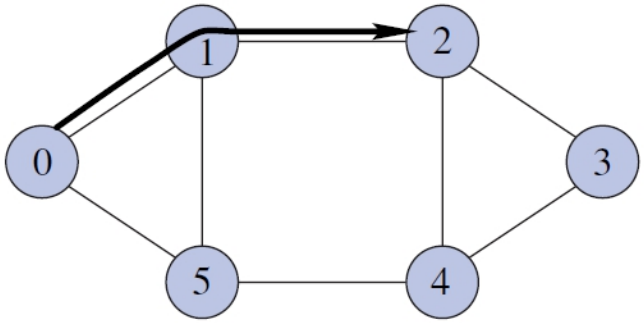


Figure 2.2: Fixed shortest-path route from Node 0 to Node 2 [2].

This approach is unarguably simple, but might lead to a critical drawback. In fact, if the topology presents a distance-wise convenient link, several pre-determined routes will span this link, and this would lead to a high number of wavelengths in usage, and hence to a potentially high blocking probability.

Fixed-Alternate Routing

The simplest routing approach that considers multiple routes is fixed-alternate routing. Given a source-destination pair, two routes from source to destination are said to be alternate if they do not share any link (Fig. 2.3).

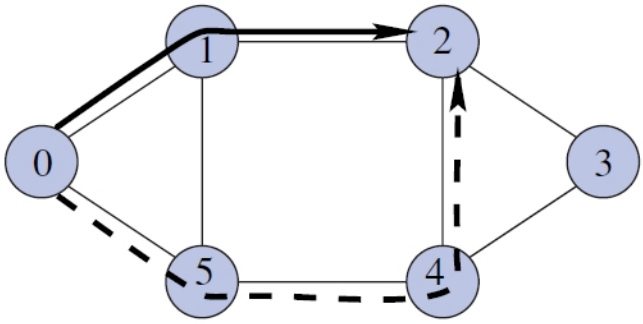


Figure 2.3: Two alternate routes from Node 0 to Node 2 [2].

Therefore, in fixed-alternate routing, each node in the network is required to maintain an own routing table containing an ordered list of off-line computed

alternate routes to each destination node. The list could be made, for example, by the shortest-path route, the shortest-path alternate route, the second shortest-path alternate route, and so on.
When a connection request arrives, the source node attempt to establish the call on each of the routes from the routing table in sequence, until a route with a valid wavelength is found, according to the selected WA approach. If no available route is found from the list, then the connection is blocked.
Although this approach presents the drawback of the routing tables storage and a larger number of computations during the DLE window, the great advantage of fixed-alternate routing is that it can significantly reduce the connection blocking probability compared to fixed routing.

Adaptive Routing

In adaptive routing, the route from a source node to a destination is chosen dynamically, depending on the network state, which is determined by the set of all connections that are currently in progress.
The idea is to route an incoming connection not on a set of pre-determined routes, but on a route which is not congested or close to being congested. A possible implementation is adaptive shortest-cost-path routing: each link in the network has a cost of k units, being k the number of lightpaths currently spanning that link. When k becomes equal to the number of available wavelengths, the link is congested and its cost is automatically set to ∞. Therefore, a minimum-cost routing algorithm is implemented; if no finite-cost route is found, then the call is blocked.
In Fig. 2.4, one can observe an adaptive route from Node 0 to Node 2: although links (1,2) and (4,2) are congested, the adaptive-routing algorithm can still establish a connection between Nodes 0 and 2, whereas the above mentioned approaches would block the call.

Indeed, by exploring all the possible paths from any source to any destination, adaptive routing results in a lower blocking probability than fixed and fixed-alternate routing. The cost to be paid for such an advantage is that this dynamic approach requires an extensive support from the control and management protocols to continuously update the routing tables at the nodes. Moreover, the average routes length will result larger compared to the other approaches, since longer paths are considered in order to bypass links congestion.

Other Routing Approaches

The ones above explained are definitely the simplest and most popular approaches as solution to the R supbroblem. However, more complex and sophisticated solutions

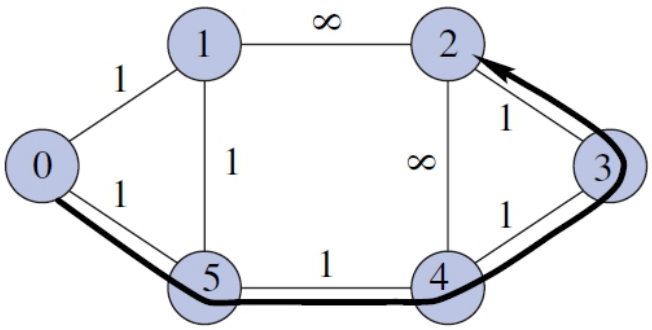


Figure 2.4: Adaptive route from Node 0 to Node 2 [2].

have been proposed in the literature. One may be interested in reading about the least-congested-path routing in [2] and the Generic Dijkstra algorithm in [7].

2.3.3 WA Subproblem

Once the R subproblem has been solved by returning a route for the incoming connection request, this route is given as input to the WA subproblem. The WA subproblem, given the route and the network current state (local or global) in terms of links occupation, determines whether the connection has been accepted (and which wavelength has been allocated), or blocked.
Since in the DLE scenario the connection requests are not known in advance, heuristic methods must be used to assign wavelengths to lightpaths. Given a fixed and ordered number of wavelengths, these heuristic methods attempt to minimize connection blocking in a sub-optimal way.
Several heuristics have been proposed in the literature to solve the WA subproblem, and this section aims at describing the most popular ones. All these heuristics are implemented as on-line algorithms and can be combined with any of the discussed routing schemes.

First-Fit WA

First-Fit (FF) is the simplest heuristic method for the WA subproblem. Given the route and the ordered list of wavelengths, the FF scheme checks if the first wavelength is available on the route. If yes, the call is accepted and the first wavelength is allocated; else, the scheme checks if the second wavelength is available on the route, and so on until an available wavelength is found. If no available wavelength is found, then the request is blocked.
FF performs well in terms of blocking probability and fairness. Moreover, it does

not introduce any communication overhead because no global knowledge is required, and hence its complexity is low.

Random-Fit WA

Given the route and the list of wavelengths, Random-Fit (RF) checks if all the wavelengths are available on the route, and therefore determines the list of feasible wavelengths. Next, it chooses randomly (usually with uniform probability) one wavelength among all the available ones.
RF does not require any global knowledge, but, compared to FF, the computation cost of this scheme is higher because it searches the entire space of wavelengths for every incoming call. Moreover, RF is outperformed by FF in terms of blocking probability.

Least-Used WA

Given the route and the current network state, the Least-Used (LU) scheme always selects the wavelengths that is the least used in the network. The LU performance is worse than RF, while also introducing additional communication overhead since global information is needed at every computation.

Most-Used WA

Most-Used (MU) is the opposite of LU since it attempts to select the most used wavelength in the network. The communication overhead, storage and cost are all similar to those in LU. However, MU significantly outperforms LU and slightly outperforms FF.

These four heuristics are the simplest and most popular in the literature. The summary is that, in terms of blocking probability, MU outperforms FF, which outperforms RF, which outperforms LU. However, one should also consider computational costs and communication overhead, which are strongly larger in MU and LU compared to FF and RF.
For a more detailed overview on heuristics results and complexity comparison, one could refer to [2] and [8]. Also, if one is interested in studying more sophisticated and complex methods to be implemented in multi-fiber networks, can still refer to [2].

Chapter 3

Machine Learning Methods and Graph Neural Networks

3.1 Introduction

3.1.1 The Increasing Interest in Machine Learning

The interest in Artificial Intelligence (AI) and, more specifically, in the area of Machine Learning (ML) has been increasing rapidly in the networking community in recent years. Much attention has been devoted to the question of whether/when traditional network protocol design, which relies on the application of algorithmic insights by human experts, can be replaced by a data-driven (i.e. ML) approach. In fact, nowadays enormous amounts of data can be retrieved from telecommunication networks, for example provided by network telemetry, quality indicators of physical signals, data traffic traces and logs, user profiling etc. Such an abundance of data has recently lead to the belief that, by leveraging the methodologies of ML, several complex networking tasks can be performed with high accuracy and with limited human intervention.
Moreover, such an increasing interest in ML methods applied to telecommunication networks is due to the research and development of a revolutionary networking principle, which is Software Defined Networking (SDN). Indeed, SDN is finally becoming a reality and offers the opportunity to re-think and build highly programmable networks. As [5] states, "thanks to SDN, global-view networked datasets comprising forwarding, performance and configuration states can be gathered and further exploited with ML/AI algorithms, offering a new range of possibilities to continuously improve how network services are provided and network resources allocated".
Therefore, SDN is a promising application field for the implementation of ML

algorithms. The main strength of this new paradigm is that different network optimization algorithms can be implemented, each targeting a different cost function. Moreover, thanks to the centrality of the control plane, it is possible to simultaneously train different ML algorithms in off-line mode, i.e. without implementing the output in the network, and only after generalizing and testing the model, this can be deployed. This procedure can be repeated any time the model needs to be retrained, following the evolution of network behavior itself.
In conclusion, SDN enables intelligent control and configuration actions in a very short time and, together with ML/AI, represents the fundamental feature of future telecommunication networks.

3.1.2 Related Work

As explained above, in the recent years the research community has applied ML methods to different traditional networking problems.
One of the first and straightforward works about data-driven routing can be found in [9]. [10] proposes an AI-based routing paradigm for Multi-Domain Optical Networks, achieving excellent accuracies and significant signaling reduction. [11] shows the interesting results achieved by a deep-reinforcement-model for routing, modulation and spectrum assignment (Deep-RMSA) in elastic optical networks (EONs). In [12], one can read about a ML-based Routing of QoS-constrained connectivity services in optical transport networks (OTNs). [13] defines a Graph-Aware Deep Learning (GADL) based intelligent routing strategy.
All the ones above mentioned are important references which have leveraged different ML models in order to solve the routing problem. From the point of view of this project, the most interesting related work can be found in [5], where Martìn et al. propose a ML-based static RWA in SDNs. Their ML model turns to be capable of solving the SLE problem by achieving high accuracies and reducing computational time up to 93% in comparison to a traditional optimization approach based on ILP. Their work can be taken as an important initial reference for ours, but it must be considered that they have applied ML to the SLE in a very small network (5 nodes), whereas, as the reader will see through this thesis, our work will be focused on applying ML methods in order to solve the DLE problem in a way larger network, thereby representing a more challenging task.
In particular, the ML methods of this project will be based on a particular type of neural networks, which is Graph Neural Networks (GNNs). A detailed overview on GNNs will be held in the next sections, but the reader can also find important references in [14] and [15].

3.2 Machine Learning Methods and Neural Networks

In the wide area of the always-on-trend AI, the most popular and researched field is definitely ML. To give a definition of such a vast discipline, ML is "the study of computer algorithms that improve automatically through experience" [16]. In fact, ML algorithms build in general a model based on sample data, known as *training data*, in order to make predictions or decisions without being explicitly programmed to do so.
The section presents a general overview on ML approaches and models, which will turn useful through the thesis to understand how a particular ML model, called Graph Neural Network (GNN), has been leveraged and applied to the dynamic RWA problem.

3.2.1 Machine Learning Approaches

The discipline of ML employs various approaches to teach machines to accomplish tasks where no fully satisfactory algorithm is available. ML approaches are traditionally divided into three broad categories, depending on the nature of the feedback available to the learning system: supervised learning, unsupervised learning and reinforcement learning [9].

Supervised Learning

The general idea of Supervised Learning (SL) is to "feed" the computer with some example inputs and the corresponding desired outputs (called *labels*), and the goal is to learn a general rule that maps any new input to the proper output. In other words, SL is a formalization of the learning process through examples, i.e. in a supervised way.
More in detail, SL algorithms build a mathematical model starting from a set of data that contains both the inputs and the desired labels. The data is known as training data, and consists of a set of training examples. Through iterative optimization of a cost function, SL algorithms learn a function that can be used to predict the output associated with new inputs. An optimal function will allow the algorithm to correctly determine the output for inputs that were not part of the training data, which is the general goal to achieve for any ML model. In fact, an algorithm that improves the accuracy of its outputs over time is said to have learned to perform that task, and hence is expected solve with the highest possible accuracy unlabeled examples, known as test data.
Types of SL tasks include classification and regression. Classification algorithms are used when the outputs are restricted to a limited set of values, whereas regression

algorithms are used when the outputs may have any numerical value within a range. As the reader will see through the next chapters, the DLE problem will be treated as a SL classification problem in which the goal is to correctly classify nodes and edges belonging to the route from source to destination.
SL finds a vast usage in all the applications where a large set of data is available, such as object and speech recognition, spam identification, face verification.

Unsupervised Learning

The Unsupervised Learning (UL) approach is exactly the opposite of SL, meaning that no labels are given to the learning algorithm, leaving it on its own to find structures in its input. The idea is to let the computer learn in absence of examples, thereby in a unsupervised way.
More in detail, UL algorithms take a set of data that contains only inputs, and hence have not been labeled, classified or categorized. The algorithm then tries to find some structures in the data, like grouping or clustering of data points. Given the absence of labeled targets and hence feedback, UL algorithms identify similarities in the data and react based on the presence or absence of such similarities in each new piece of data.
In general, UL can perform tasks that are more complex than the SL ones. Among the most important ones, one can find data clustering, network analysis, market research and density estimation.

Semi-supervised Learning

Semi-supervised Learning (Semi-SL) falls in the middle between SL and UL. In fact, Semi-SL algorithms learn by means of a training data which is made in part by labeled examples and in part by unlabeled examples.
Such an approach is useful when some of the training examples are missing their labels. Moreover, evidence has shown that unlabeled data, when used in conjunction with a small amount of labeled data, can produce a considerable improvement in learning accuracy.

Reinforcement Learning

In Reinforcement Learning (RL), the ML algorithm relies on an agent program which interacts with a dynamic environment in which it must perform a given task. As it navigates its problem space, the program is provided feedback that's analogous to rewards, which it tries to maximize.
In particular, time is divided into discrete time slots. At the beginning of each time slot, the agent observes the current state of the environment and selects an action from a fixed set of actions. Once the agent chooses the action, the state of

the environment changes and the agent receives a reward signifying how good/bad the action was. Time slot by time slot, the agent is supposed to learn by the received rewards and regulate its actions depending on the current state in order to maximize the final reward.
The RL approach can be used to perform tasks which can be learned by this reinforcement procedure, such as driving a vehicle or playing a game against an opponent.

Other Approaches

Other learning approaches different from these three categories are Self-Learning, Feature Learning, Sparse, Dictionary Learning, Anomaly Detection, Robot Learning, Association Rules Learning. The reader can find more about these approaches in [17].

3.2.2 Machine Learning Models

Performing ML involves creating a model, which is first trained on some training data and then can be tested on some test data. Although Neural Networks are definitely the most popular model associated with ML, various types of models have been researched and used for ML systems.

Artificial Neural Networks

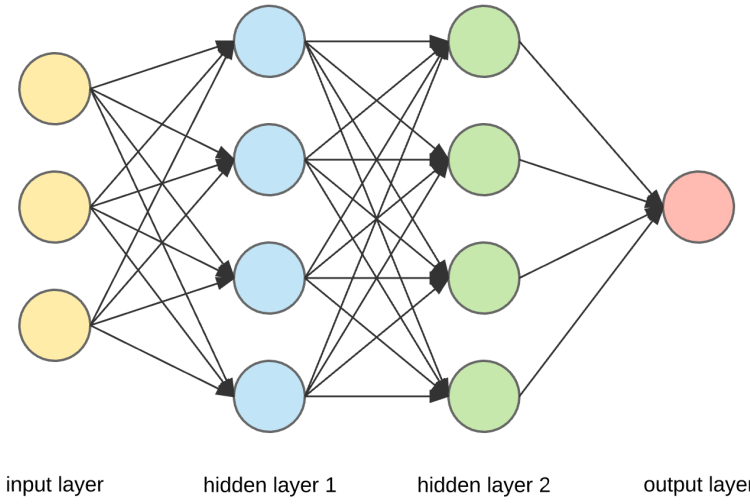


Figure 3.1: A simple artificial neural network.

Artificial neural networks (ANNs), or simply neural networks (NNs), are a ML

computing model inspired by the biological neural networks that constitute human and animal brains. Such systems turn out to be extremely reliable when it comes to learn performing a new task not on the basis of some pre-programmed rules, but just on the basis of some available data.
A NN is based on a collection of processing units called artificial neurons or *perceptrons*. Such neurons are typically represented by nodes in a graph, and are connected to each other via directed edges (Fig. 3.1). Each connection, like the synapses in a biological brain, can transmit a signal (a real number) between neurons. A neuron that receives a signal, processes it and can transmit it to other neurons connected to it.
In particular, a neuron is generally connected to multiple neurons, and hence receives multiple input values. Then, the neuron computes its output value by means of a specific non-linear function, called *activation function*, of the weighted sum of its inputs (Fig. 3.2).

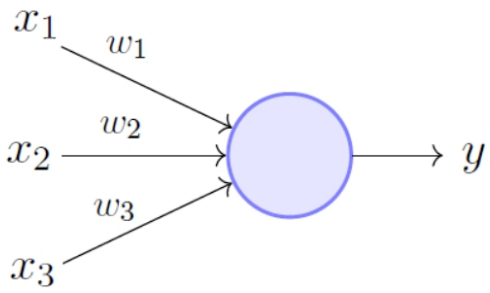


Figure 3.2: A perceptron receiving three input values.
x_i is the input value transmitted on the $i-th$ edge, w_i is the current weight of the $i-th$ edge, y is the perceptron output obtained by applying the activation function to $\sum_i w_i x_i$

Given a *cost function* representing the difference between NN output and target labels, neurons and edges are assigned some initial weights, whose values change as the learning phase proceeds in order to minimize such a cost function.
Typically, neurons are aggregated into layers. The first NN layer is called *input layer*, whereas the last layer is called *output layer*. All the layers in the middle are referred to as *hidden layers*. In Fig. 3.1 one can observe a NN made of four layers. Therefore, during the training phase, signals travel from the input layer to the output layer, possibly after traversing multiple hidden layers.
The output layer is the one which returns an output, a prediction or classification based on the type of problem. Such a outcome is compared to the corresponding target label and, through the so-called *back-propagation algorithm* [18], the NN

adjusts the nodes and edges weights in order to minimize the cost function, and hence to increase the accuracy. Such a process is repeated iteratively until the training phase is terminated.
Since the first researches about this still young AI field, NNs have been used on a variety of tasks, including computer vision, speech recognition, machine translation, social network filtering, playing board and video games and medical diagnosis.
Based on their internal architecture and learning algorithm, NNs can be classified as belonging to various categories, such as deep NNs, convolutional NNs (CNNs), recurrent NNs (RNNs) etc. In particular, a deep NN is a NN which consists of multiple hidden layers and is sometimes referred to as Multilayer Perceptrons (MLPs) network. For a deep insight on these different NNs structures, the reader may refer to [18].
For the purposes of this project, the most important type of NN is a graph-structured NN (GNN), which will be discussed in the next section.

Decision Trees

Decision tree is a predictive modeling approach used in ML. Such a model uses a decision tree as a predictive model to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves).
A fundamental distinction is done. Tree models where the target variable can take a discrete set of values are called *classification trees*; in these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Decision trees where the target variable can take continuous values are called *regression trees*.
The most remarkable application is decision analysis, where a decision tree can be used to visually and explicitly represent decisions and decision making.

Support Vector Machines

Support Vector Machines (SVMs) are a set of related supervised learning methods used for classification and regression. Given a set of training examples, each marked as belonging to one of two categories, the SVM relies on a non-probabilistic, binary, linear classifier training algorithm which builds a model that predicts whether a new example falls into one category or the other.

Other ML models that can be found in the literature are regression analysis, Bayesian networks and genetic algorithms [16].

3.3 Graph Neural Networks

3.3.1 A Multitude of Graph-Structured Tasks

Among all the real-world tasks, several of these are conveniently formulated as tasks over graph-structured inputs, such as navigation, web search, protein folding, and game-playing. Over the years, theoretical computer science has successfully discovered effective and highly influential algorithms for many of these tasks, but many problems are still considered intractable from this perspective. In fact, although some problems can be easily solved by well-known polynomial time algorithms such as *shortest path* and *sorting*, there exist more complex and non-polynomially-solvable problems such as *travelling salesman*, *boolean satisfability* and, as already explained, the *RWA* problem.
Despite classical ML methods have been applied to solve these graph-based tasks over the years, it has been more recently that a new idea has become a trend in the ML area. Such an idea consists of leveraging the problems' graph-based structure and hence building a graph-based ML model that could get advantages from it. In fact, NNs that operate on graphs, and structure their computations accordingly, have been developed and explored extensively in the last decade, and have grown rapidly in scope and popularity. Such a new concept of NNs based on graphs are referred to as Graph Neural Networks (GNNs). GNNs are proposed to collectively aggregate information from a graph structure, and can model input and/or output consisting of elements and their dependency.

3.3.2 Definition of Graph

Graphs are a kind of data structure which models a set of objects and their relationships. Objects are represented by nodes, relationships are represented by edges between nodes.
Although there is a universal agreement on the concept of graph, in the literature there exist several formal definitions. Within this project, the definition provided by [19] will be used as reference. Formally, a *graph* is defined as a 3-tuple $G = (\mathbf{u}, V, E)$, where:

- $\mathbf{u}$ is the graph global attribute. In a networking context, for instance, $\mathbf{u}$ might represent the network global utilization.
- V is the set of nodes and has cardinality N^v; in particular, $V = \{v_i\}_{i=1:N^v}$, where each v_i is a node's attribute. In a networking scenario, for instance, V might represent each router/switch with attributes for position.
- E is the set of edges and has cardinality N^e; in particular, $E = \{(e_k, s_k, r_k)\}_{k=1:N^v}$, where each e_k is the edge's attribute, r_k is the index of the receiver node, and

s_k is the index of the sender node. In a networking scenario, for instance, E might represent the presence of links between routers/switches with attributes for distance, available resources etc.

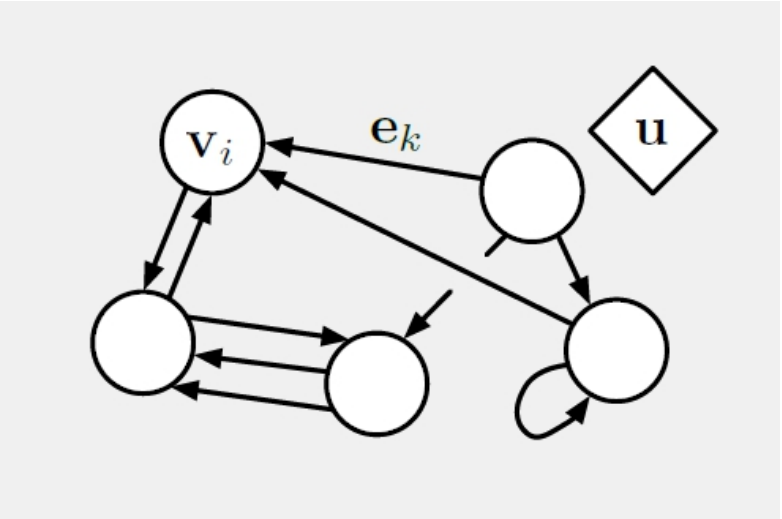


Figure 3.3: Graph representation according to the formal definition of [19].

In Fig. 3.3 one can observe a simple graph represented according to the formal definition given above.
In general, here the term *graph* denotes a directed, attributed multi-graph with a global attribute. To be more precise, these terms are defined as:

- *Directed*: one-way edges, from a sender node to a receiver node.
- *Attribute*: properties that can be encoded as a vector, set, or even another graph.
- *Attributed*: edges and nodes have attributes associated with them.
- *Global attribute*: a graph-level attribute.
- *Multi-graph*: there can be more than one edge between nodes, including self-edges.

The ones above are general properties, and, as the reader will see in the next chapters, not all of them hold for our graph data.

3.3.3 GN Block

The main unit of computation in a GNN is the *Graph Network Block* (GN block). The GN block is a graph-to-graph module which takes a graph as input, performs computations over the structure, and returns a graph as output.
In the following, an insight about the internal structure of the GN block will be presented, as well as the computational steps which are performed by it. Once these topics are discussed, it will be easier to understand how the GN blocks configuration can define a specific GNN architecture.

Internal Structure of a GN Block

A GN block contains three *update* functions, ϕ, and three *aggregation* functions, ρ. Such functions are defined as:

- $e'_k = \phi^e(e_k, v_{rk}, v_{sk}, \mathbf{u})$
- $\overline{e}'_i = \rho^{e\to v}(E'_i)$
- $v'_i = \phi^v(\overline{e}'_i, v_i, \mathbf{u})$
- $\overline{e}' = \rho^{e\to u}(E')$
- $\mathbf{u}' = \phi^u(\overline{e}', \overline{v}', \mathbf{u})$
- $\overline{v}' = \rho^{v\to u}(V')$

where $E'_i = \{(e'_k, r_k, s_k)\}_{r_k=i, k=1:N^e}$, $V' = \{v'_i\}_{1:N^v}$, and $E' = \bigcup_i E'_i = \\ = \{(e'_k, r_k, s_k)\}_{k=1:N^e}$.

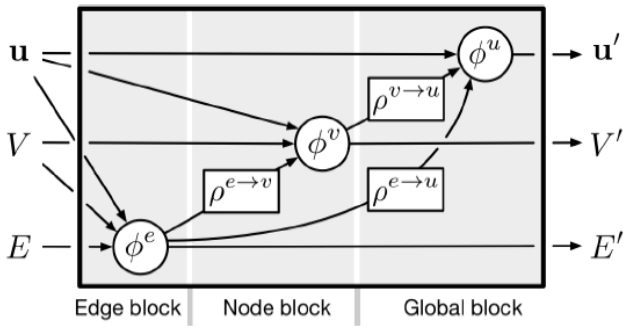
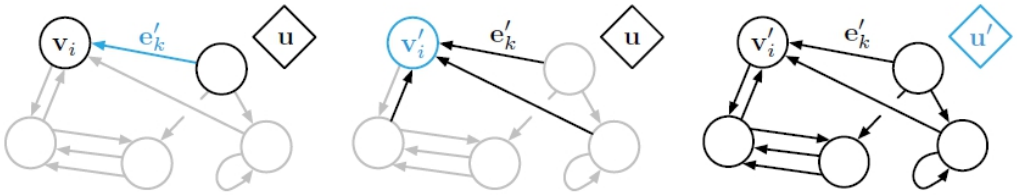


Figure 3.4: Full GN Block [19].

More in detail, the ϕ^e function is mapped across all edges to compute per-edge updates, the ϕ^v function is mapped across all the nodes to compute per-nodes updates, and the ϕ^u function is applied once as the global update. Each of the ρ function takes a set as input, and reduces it to a single element which represents the aggregated information. Crucially, the ρ functions must be invariant to permutations of their inputs, and should take as input variable numbers of arguments. For instance, these can be element-wise summation, mean, maximum, etc. For a graphic representation of the GN block internal structure, one can observe Fig. 3.4, which shows a full GN block with its update and aggregation functions.

Computational Steps within a GN Block

When a graph G is provided as input to a GN block, the computations proceed from the edge, to the node, to the global level.
The computational process of a GN block is resumed in Algorithm 1 (Fig 3.6); moreover, for a clearer insight on such algorithm one may refer to Fig. 3.5, which graphically shows which graph elements are involved in each specific computation.



(a) Edge update (b) Node update (c) Global update

Figure 3.5: Updates in a GN block. Blue indicates the element that is being updated, whereas black indicates other elements which are involved in the update [19].

The computational steps of Algorithm 1 are sequentially described in detail as follows:

1. **Step 1** ϕ^e is applied per edge, with arguments $(e_k, v_{rk}, v_{sk}, \mathbf{u})$, and returns e'_k. The set of resulting per-edge outputs for each node, i, is $E'_i = \{(e'_k, r_k, s_k)\}_{r_k=i, k=1:N^e}$. And $E' = \bigcup_i E'_i = \{(e'_k, r_k, s_k)\}_{k=1:N^e}$ is the set of all per-edge outputs.

2. **Step 2** $\rho^{e\to v}$ is applied to E'_i, and aggregates the edge updates for edges that project to vertex i, into $\overline{e}'_i$, which will be used in the next step.

Algorithm 1 Steps of computation in a full GN block.

function GRAPHNETWORK(E, V, $\mathbf{u}$)
  for $k \in \{1 \dots N^e\}$ **do**
    $\mathbf{e}'_k \leftarrow \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$ ▷ 1. Compute updated edge attributes
  end for
  for $i \in \{1 \dots N^n\}$ **do**
    let $E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i,\ k=1:N^e}$
    $\bar{\mathbf{e}}'_i \leftarrow \rho^{e\to v}(E'_i)$ ▷ 2. Aggregate edge attributes per node
    $\mathbf{v}'_i \leftarrow \phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$ ▷ 3. Compute updated node attributes
  end for
  let $V' = \{\mathbf{v}'\}_{i=1:N^v}$
  let $E' = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e}$
  $\bar{\mathbf{e}}' \leftarrow \rho^{e\to u}(E')$ ▷ 4. Aggregate edge attributes globally
  $\bar{\mathbf{v}}' \leftarrow \rho^{v\to u}(V')$ ▷ 5. Aggregate node attributes globally
  $\mathbf{u}' \leftarrow \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u})$ ▷ 6. Compute updated global attribute
  return $(E', V', \mathbf{u}')$
end function

Figure 3.6: Algorithm 1 [19].

3. **Step 3** ϕ^v is applied to each node i to compute an updated node attribute v'_i. The set of resulting per-node outputs is $V' = \{v'_i\}_{i=1:N^v}$.

4. **Step 4** $\rho^{e\to u}$ is applied to E', and aggregates all the edge updates into $\overline{e}'$, which will be used in step 6.

5. **Step 5** $\rho^{v\to u}$ is applied to V', and aggregates all the node updates into $\overline{v}'$, which will be used in step 6.

6. **Step 6** ϕ^u is applied once per graph, and computes an update for the global attribute **u'**.

The GN block is the fundamental computational unit of a GNN. Given its structure and functionalities as assimilated, the next section will offer an overview of how to design a specific GNN architecture starting from the GN block.

3.4 Design of a GNN Architecture

The GNN framework can be used to implement a wide variety of architectures, each in accordance to the design principles which are discussed in this section. In particular, for GNNs, three design principles are distinguished: flexible graph representation, configurable GN block structure and composable multi-block architecture.
Based on the design choices, one can distinguish among various GNN architectures. For the purposes of this project, the most relevant is the so-called Message-Passing Neural Network (MPNN), which is explained in detail in subsection 3.4.2.

3.4.1 Graph Representation

GNNs support a flexible graph representation in two ways: first, in terms of the representation of the attributes; second, in terms of the structure of the graph itself.

Attributes Representation

The edge, node and global attributes of a GN block can be represented according to different formats. In deep learning applications, real-valued vectors and tensors are the most common. However, other data structure such as sequences, sets, or even graphs can be used.
For each GN block within a broader architecture, the edge and node outputs typically correspond to lists of vectors or tensors, one per edge or node, and the global vector output corresponds to a single vector or tensor. This allows a GN block's output to be passed to other deep learning building blocks such as MLPs, CNNs, and RNNs.
The output of a GN block can also be focused on just one of the above-mentioned attributes. In particular:

- An *edge-focused* GNN uses the edges attributes as output.
- A *node-focused* GNN uses the nodes attributes as output.
- A *graph-focused* GNN uses the global attribute as output.

A GNN can also belong to more than one of such categories, by using a mix-up of different attributes as output. As it will be discussed in Chapter 4, our GNN framework represents attributes as tensors and is both edge and node-focused.

Graph Structure

Beside the attributes, another important factor in the design of a GNN is the input data graph structure. In fact, depending on the structure representation, two scenarios might arise: in the first one, the input explicitly defines the relational structure (e.g. social networks, physical systems, chemical graphs), whereas in the second one, the relational structure is not explicit, but must be inferred or assumed (e.g. visual scenes, text corpora, programming language source code).
As it will be discussed in Chapter 4, our GNN framework is tested with an input graphs obtained from a real-world network, and hence belongs to the first scenario.

3.4.2 GN Block Configurations

The structure and functions within a GN block can be configured in different ways, each building a specific GNN architecture. The GN block configuration offers flexibility in what information is made available to its functions, as well as how output edge, node and global updates are produced.
In particular, each update function ϕ must be implemented with a specific function, whose argument signature determines what information in the space $(E, V, \mathbf{u})$ it requires as input. Typical implementations of ϕ functions are MLPs, CNNs and RNNs. For what concerns the implementation of the ρ functions, instead, typical choices are element-wise summation, average and max/min.
In the following, the reader can find an overview on the most popular GNNs architectures.

Message-Passing Neural Network (MPNN)

A Message-Passing Neural Network (MPNN) is a particular implementation of GNN that relies on an iterative message-passing algorithm to propagate information between the nodes of the graph. The GN block configuration of a MPNN can be observed in Fig. 3.7.

The model contains two phases, a message passing phase and a readout phase. According to the GN block definition of the previous section, a MPNN bases its computations on the following functions:

- The function ϕ^e does not take $\mathbf{u}$ as input and is called message function.
- the function $\rho^{e \to v}$ is implemented as an element-wise summation.
- the function ϕ^u, called readout function, does not take $\mathbf{u}$ or E' as input, and thus $\rho^{e \to u}$ is not required.

A visual representation of message-passing steps can be observed in Fig 3.8.

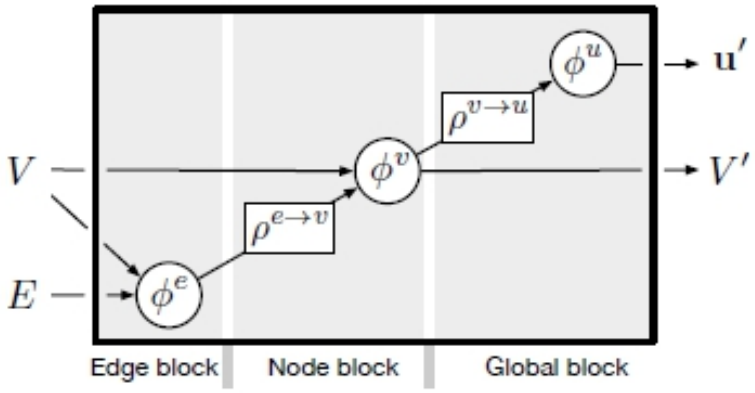


Figure 3.7: GN block configuration in a MPNN [19].

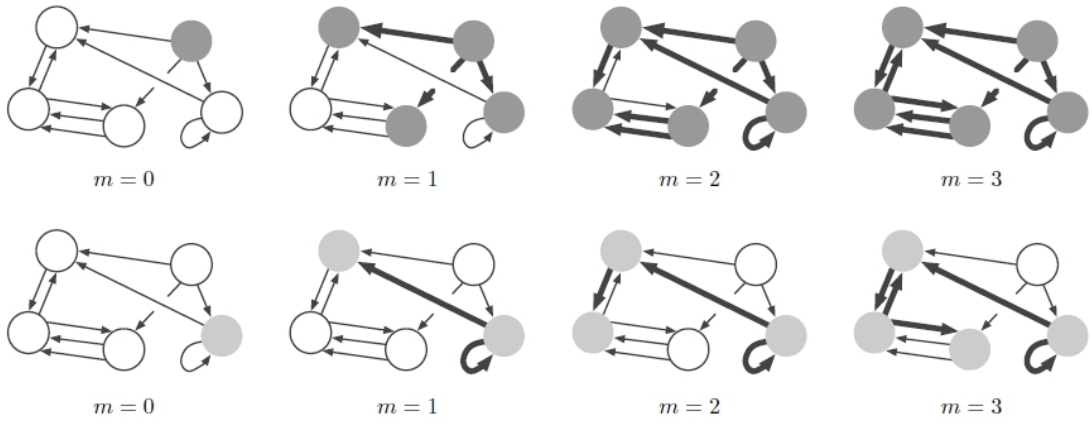


Figure 3.8: Example of message-passing. Each row highlights the information that diffuses through the graph starting from a particular node. Notice that the information that a node has access to after m steps of propagation is determined by the set of nodes and edges that are at most m steps away [19].

Non-Local Neural Network (NLNN)

A Non-Local Neural Network (NLNN) is a particular model of GNN which is based on the concept of non-local mean operation. A non-local operation computes the response at a position as a weighted sum of the features at all positions. In fact, in a NLNN each node update is based on a weighted sum of the node attributes of its neighbors, where the weight between a node and one of its neighbors is computed by a scalar pairwise function between their attributes. Therefore, a NLNN only

predicts node output attributes. The GN block configuration of a NLNN can be observed in Fig. 3.9.

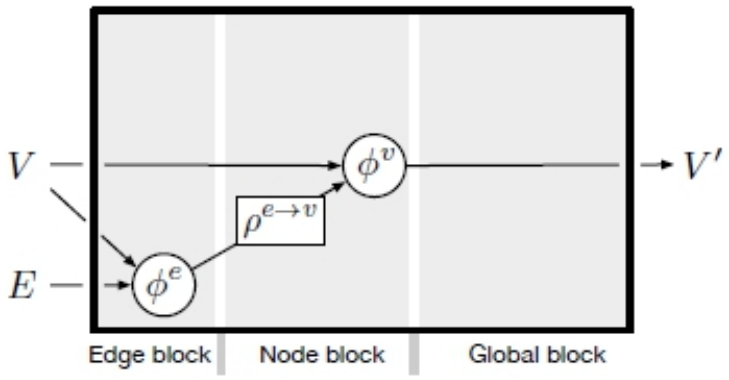


Figure 3.9: GN block configuration in a NLNN [19].

Other GNN Architectures

The flexibility given by the GN block configuration allows to model a wide variability of GNN models. In the literature, one can find defined architectures such as Relation networks, Deep sets, PointNet, CommNet, Interaction Networks and others. For an overview on the ideas behind these models, the reader is invited to refer to [19] and [15].

3.4.3 Composable Multi-Block Architectures

The third design principle of GNNs is the possibility of constructing complex architectures by composing GN blocks.
In the previous section, the GN block has been defined as a computational unit which takes a graph with edge, node and global attributes as input, and returns a graph with the same constituent elements as output. This graph-to-graph input/output interface makes possible that the output of one GN block can be passed as input to another one, even if their internal configurations are different. Formally, two GN blocks, GN_1 and GN_2, can be composed as $GN_1 \circ GN_2$ by passing the output of the first as input to the second: $G' = GN_2(GN_1(G))$. In general, an arbitrary number of GN blocks can be composed into the so-called GN_{core}, as shown in Fig 3.10.

One of the most common GNN architecture design is the *encode-process-decode* configuration, which is the one implemented in our framework (see Chapter 4). Such an architecture can be observed in Fig. 3.11.

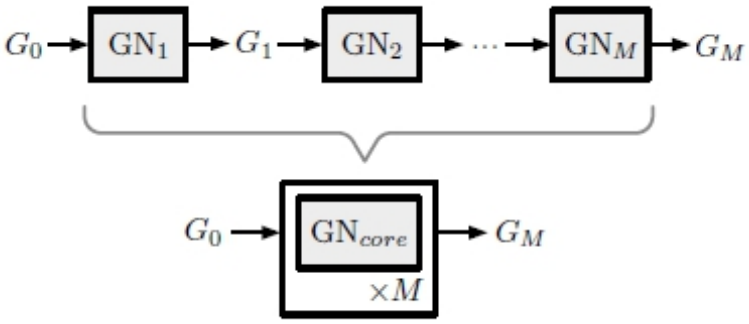


Figure 3.10: Composition of GN blocks. [19].

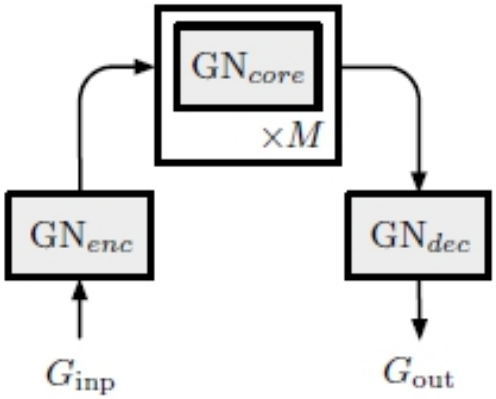


Figure 3.11: Encode-process-decode architecture [19].

In this configuration, an input graph G_{inp} is transformed into a latent representation G_0 by an encoder block, GN_{enc}. Next, a core block, GN_{core} is applied M times to return G_M, in an iterative way analogous to the message-passing principle. Finally, G_M is decoded by a decoding block, GN_{dec}, which returns the final output G_{out}.
Other common composed architectures from literature are the recurrent GN architecture and the graph skip connections configuration. For more details, the reader is invited to refer to [19].

Chapter 4

GNN Framework for the RWA Problem

The previous chapters have presented in detail the RWA background and an overview on ML concepts, with a specific focus on GNNs. Now, it finally comes the time to merge the two topics together, and hence try to leverage a GNN architecture in order to solve the RWA problem.
This idea has proved achievable by means of the *GNN framework for the RWA problem*, which is thoroughly presented in this chapter. Such a framework and all the tools that have been implemented for the purposes of the project, have been developed starting from the Graph Nets library released by [19]. This is an open-source library that builds GNNs in Python/TensorFlow/Sonnet and allows to train GNNs over graph-structured data. The software includes a demo on the shortest path problem, which was used as starting point to build our RWA-solving GNN-based framework.
If the reader is interested, the original Graph Nets library can be found at **github.com/deepmind/grap_nets**.

4.1 Graphs Data Representation

4.1.1 Input Graph Data

The first step for the framework comprehension is a clear presentation of the input graph data which are used to generate the RWA training and test data for the GNN.
As deeply discussed in Chapter 2, given a network topology and a list of available wavelengths, the DLE problem consists in finding a route and possibly assigning a wavelength to any incoming connection request. Therefore, the fundamental input

data unit for the RWA problem is the incoming connection request, simply defined by a *source-destination* pair. However, since GNNs work on graph-structured data, such a concept of call request must be moved to an input graph-based representation.
Therefore, within this framework, an incoming connection request on a given topology is represented by a graph data which is a 2-tuple $G = (V, E)$, where V is the set of nodes and E is the set of bidirectional edges. Notice that, according to the general definition of graph given in the last chapter, the global attribute $\mathbf{u}$ is missing, since meaningless in our context. Moreover, here edges are bidirectional, but the representation is still coherent with the previous one because a bidirectional edge can be seen as the union of two directed edges.
The available number of wavelengths initially available on any edge is W, and we refer to an ordered list of available wavelengths $[\lambda_1, \lambda_2, ..., \lambda_W]$.
In particular, every node $v \in V$ is characterized by the following attributes:

- *source*: boolean equal to $True$ if the node is the source of the connection request, $False$ otherwise.
- *destination*: boolean equal to $True$ if the node is the destination of the connection request, $False$ otherwise.

In any graph data within this framework, no more than one node can be the source and no more than one node can be the destination; moreover, no node can be source and destination at the same time.
Every bidirectional edge $e \in E$ is characterized by the following attributes:

- $node_A$: a node $a \in V$, representing one of the two extremes of the edge.
- $node_B$: node $b \in V$, representing the other extreme of the edge.
- *distance*: a real number representing the physical length of the edge.
- *available wavelengths*: an ordered list of the currently available wavelengths on the edge. In detail, the list is a one-dimensional array of $W + 1$ binary entries $[0, \lambda_1^e, ..., \lambda_W^e]$ where:
  - the 0-th entry is always equal to 0.
  - the i-th entry λ_i^e is equal to 1 if the i-th wavelength λ_i is currently available on the edge e, 0 otherwise.

The utility of the 0-th entry will be clearer in the next sections. Notice that, since self-edges are not here considered, no edge can have attributes $node_A = node_B$. Moreover, edges must be intended as bidirectional: the edge (a, b) enables direct

communication between nodes a and b in a bidirectional fashion, and hence a and b can be interchanged.
A graphical example of an incoming connection request on a small topology with $W = 4$ initially available wavelengths can be observed in Fig. 4.1, whereas its nodes and edges attributes can be respectively found in Table 4.1 and Table 4.2.

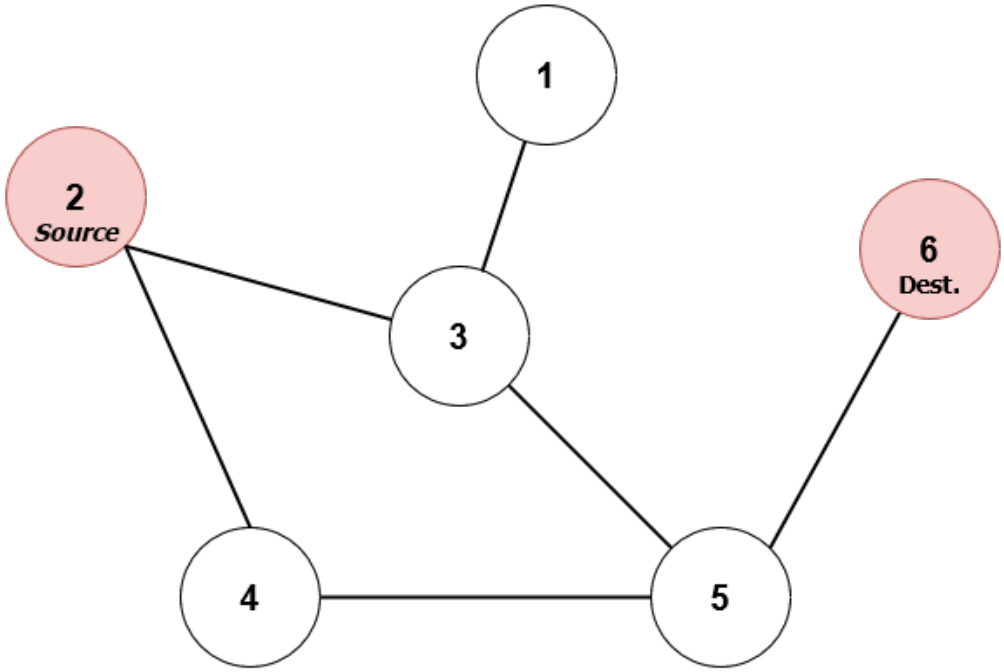


Figure 4.1: Graph representation of an input incoming connection request from Node 2 to Node 6.
Nodes and edges attributes can be respectively found in Table 4.1. and Table 4.2

Node	*source* attribute	*destination* attribute
1	$False$	$False$
2	$True$	$False$
3	$False$	$False$
4	$False$	$False$
5	$False$	$False$
6	$False$	$True$

Table 4.1: Nodes attributes of the input graph in Fig. 4.1.
As turns out from the attributes, Node 2 is the source node and Node 6 is the destination node.

Edge	$node_A$	$node_B$	$distance$	$available\ wavelengths$
(1,3)	1	3	1.50	[0,1,1,1,1]
(2,3)	2	3	1.80	[0,1,0,1,1]
(2,4)	2	4	2.10	[0,1,0,1,1]
(3,5)	3	5	2.20	[0,1,1,1,1]
(4,5)	4	5	1.90	[0,0,1,1,1]
(5,6)	5	6	2.00	[0,0,1,1,1]

Table 4.2: Edges attributes of the input graph in Fig. 4.1.
As turns out from the attributes: Edges (1,3) and (3,5) have all wavelengths available; Edges (4,5) and (5,6) have wavelength λ_1 which is allocated, all the others available; Edges (2,3) and (2,4) have wavelength λ_2 which is allocated, all the others available.

4.1.2 Target Graph Data

The input graph data representation has been just discussed. However, during the training phase, our GNN model needs to be fed with some input data and compare the predictions with the corresponding output data, also called labeled data or target data. Therefore, given any input graph, a process that generates the corresponding target graph is needed.
In particular, the framework uses a GNN which is both edge and node-focused, i.e., it labels nodes and edges of the input graph depending on whether they are either part of the solution or not. To find a solution for the input graph, the RWA problem is decomposed in R subproblem and WA subproblem. The R subproblem can be solved by any of the routing approaches discussed in Chapter 2, and the WA subproblem can be solved by any of of the heuristics summarized in the same chapter. Such a sequence of two algorithms can return:

- a route r made by p links and a wavelength λ_i allocated on r, if the call is accepted; in particular, $r = [(s_1, d_1), ..., (s_p, d_p)]$, where s_i and d_i are respectively the start and the end of the *i-th* link of the route.

- a boolean $False$, if the call is blocked.

If the call is accepted, all the nodes and edges forming the route are said to be part of the solution; if the call is blocked, there are no nodes or edges included in the solution.
For instance, if one chooses to solve the input graph of Fig. 4.1 by implementing the fixed routing with Dijkstra's algorithm followed by the First-Fit heuristics, the solution route and wavelength can be observed in Fig 4.2.

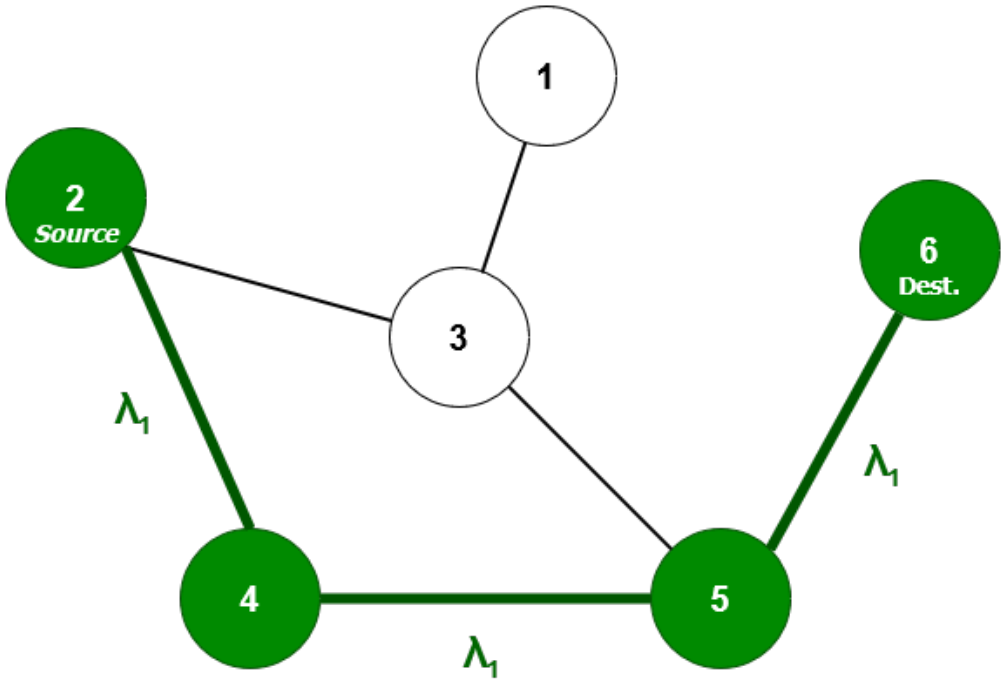


Figure 4.2: Dijkstra & First-Fit solution for the input graph of Fig.4.1. The solution route is [(2,4),(4,5),(5,6)] with allocated wavelength λ_1.

Next, once the solution nodes and edges are found, they must be labeled in order to generate the target graph. Within this framework, nodes are labeled with 2-entries one-hot vectors and edges are labeled with $(W+1)$-entries one hot vectors. In particular, any node $v \in V$ is labeled:

- with label [1,0], if v is not part of the solution.
- with label [0,1], if v is part of the solution.

Any edge $e \in E$ is labeled:

- with label [1,0,...,0], if e is not part of the solution.
- with label [0,...,1,...,0], if e is part of the solution with allocated wavelength λ_i, where i is the index of the 1-entry.

The labels for nodes and edges of the solution shown in Fig. 4.2 can be respectively observed in Table 4.3 and Table 4.4.
Notice that the length of the edge labels is the same of the input graph's edge attributes, due to the first entry being always equal to zero in the edges attributes, as explained previously.

Node	Label
1	[1,0]
2	[0,1]
3	[1,0]
4	[0,1]
5	[0,1]
6	[0,1]

Table 4.3: Nodes labels for the solution of Fig. 4.2

Edge	Label
(1,3)	[1,0,0,0,0]
(2,3)	[1,0,0,0,0]
(2,4)	[0,1,0,0,0]
(3,5)	[1,0,0,0,0]
(4,5)	[0,1,0,0,0]
(5,6)	[0,1,0,0,0]

Table 4.4: Edges labels for the solution of Fig. 4.2

4.1.3 Calls Generation Process

The last paragraphs have presented a detailed overview about the data representation of a single input graph and the corresponding target graph within the GNN framework. However, in order to let the GNN perform the training phase, one needs multiple data, i.e., a list of input graphs and a list of the corresponding target graphs.
To generate such a multitude of data, the framework relies on a simulation arrival process which generates calls requests arrivals during a time window T. In particular, if we fix the number of calls to generate as n, the process returns a list of n calls $[call_1, call_2, ..., call_n]$, each in the form:

$$call_i = (t_0^i, t_1^i, s, d)$$

where t_0^i is the arrival time, t_1^i is the termination time, s is the source node of the connection, and d is the destination node of the connection. Therefore, during the simulation window T, at any time a new call can arrive, or an active call can terminate.
If the arriving call is accepted, it is allocated a route and a wavelength for all its duration; if the arriving call is blocked, it is dropped and no resource is allocated.

Such a generation process requires some constraints:

- No call with $s = d$ can arrive.
- If a call with $s = s_1$ and $d = d_1$ is active, then, for all its duration, no other call with $s = s_1$ and $d = d_1$ or $s = d_1$ and $d = s_1$ can arrive.

Within our framework, the calls arrival process is modeled by a Poisson process with intensity λ, i.e., any new call is expected to arrive, on average, every $\frac{1}{\lambda}$ time units. When the time for a new call arrival is determined, the source s and destination d are uniformly chosen from V, always respecting the two constraints above explained.
The call duration, instead, is modeled by an exponential distribution with parameter μ, i.e., every accepted call is expected to last, on average, μ time units. Therefore, if the time window is quite long and the blocking probability is negligible, the network is expected to host, on average, $\lambda\mu$ active calls at any time.
Therefore, by means of the methods explained previously, any incoming call in the form $call_i = (t_0^i, t_1^i, s, d)$ can be moved to a graph representation and hence generate an input graph; next, by means of the RWA algorithms, from the input graph a target graph is obtained. In conclusion, to create a batch of n calls to feed the GNN, the framework generates n input graphs and n corresponding target graphs, representing the sequence of arriving calls in the simulation time window. For instance, a simulation of arrival calls, modeled by $\lambda = 5$ and $\mu = 2$, and the corresponding solutions (Dijkstra & First-Fit) for the graph of Fig 4.1. (4 wavelengths initially available) can be observed in Table 4.5.

Call ID	Call input data	Solution
1	(0.05,3.21,2,6)	[(2,4),(4,5),(5,6)], λ_1
2	(0.33,2.89,1,6)	[(1,3),(3,5),(5,6)], λ_2
3	(0.41,4.01,1,4)	[(1,3),(3,2),(2,4)], λ_3
4	(0.86,1.65,2,1)	[(2,3),(3,1)], λ_1
5	(1.12,4.34,5,2)	[(5,3),(3,2)], λ_4
6	(1.36,3.18,4,3)	[(4,2),(2,3)], λ_2
7	(1.61,2.98,3,2)	*blocked*

Table 4.5: Example of calls sequence simulation for the graph in Fig. 4.1.
Notice that the 7^{th} call has been blocked because on the link (3,2) all the wavelengths are already allocated.

4.2 The MPNN Model

4.2.1 Training Phase: a Supervised Approach

The GNN framework builds a particular architecture of GNNs, the MPNN, which has been thoroughly discussed in the previous chapter. Before seeing how the MPNN is internally structured, it is more relevant to understand the general approach of our framework.
In fact, the framework builds a MPNN which is trained following a SL approach. As deeply explained in Chapter 3, the idea of SL is to teach the NN how to perform a new task by providing various labeled examples. By iteratively comparing its predictions with the labels, the NN adjusts its weights in order to minimize a cost function and becomes better and better in performing the assigned task.
The MPNN that tries to learn how to solve the RWA problem is trained according to this principle. In particular, Fig. 4.3 shows the block diagram of one iteration of training for our MPNN.

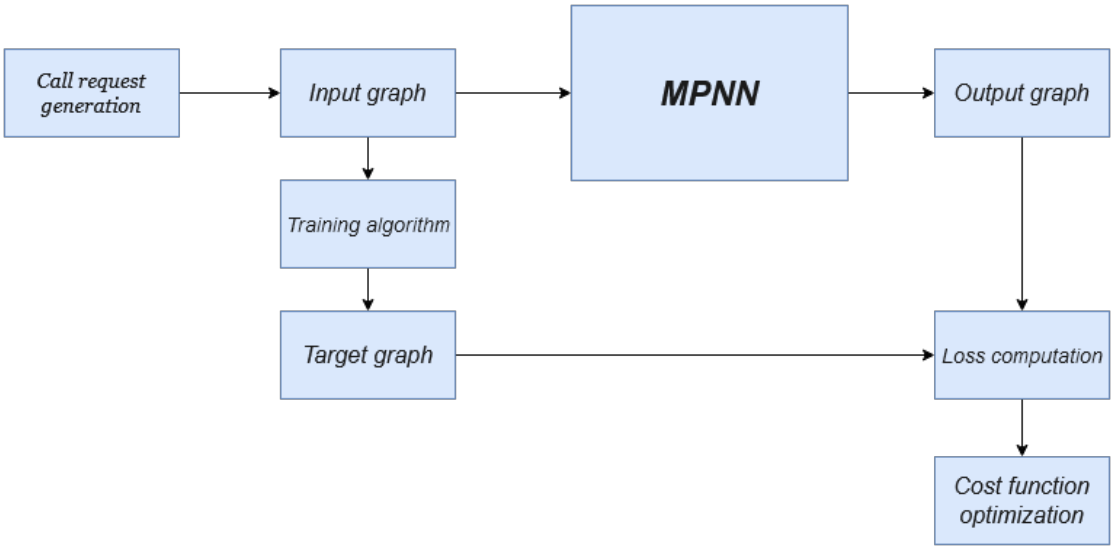


Figure 4.3: Block diagram of the MPNN training phase for the RWA problem.

As one can see, the first step is having an incoming connection request. This is transformed in the input graph representation seen in last section, and is given as input to the MPNN. In parallel, the input graph is solved by a specific choice of R algorithm followed by a WA heuristic. Such a sequence of algorithms is called *training algorithm*. The result is the corresponding target graph, with nodes and edges labeled according to the format discussed in the last section.
When the MPNN terminates its computations, it returns a labeled output graph. Such an output graph is compared with the corresponding target graph in order to

compute the loss: if the two graphs are labeled in a similar way, the loss is small, i.e. the MPNN returned a good prediction; id the two graphs are labeled in a non similar way, the loss is large, i.e. the MPNN returned a bad prediction. The loss computation is needed to adjust the MPNN's weights in order to minimize its cost function. As such a cost function decreases, the MPNN is improving the "quality" of its output graphs, and hence is getting better in solving the RWA problem. Such a training process is repeated for every iteration of the training phase.
The one above explained is the training phase of one single iteration on one single input graph. However, every iteration typically works on a *batch* of input graphs, as later explained. Moreover, the training process can be regulated by a set of *hyperparameters*, including the batch size. All the building blocks and the hyperparameters are discussed in the following.

4.2.2 The MPNN Internal Structure

Since the previous section has already presented a detailed overview about the call requests generation, the input graphs and the target graphs, the first block of the Fig 4.3 to be analyzed is the MPNN.
The MPNN within our framework is implemented according to an *Encode-process-decode* architecture. This means that the model is composed by three blocks in sequence, which are encoder, core, and decoder, followed by a simple post-processing phase (Fig. 4.4).

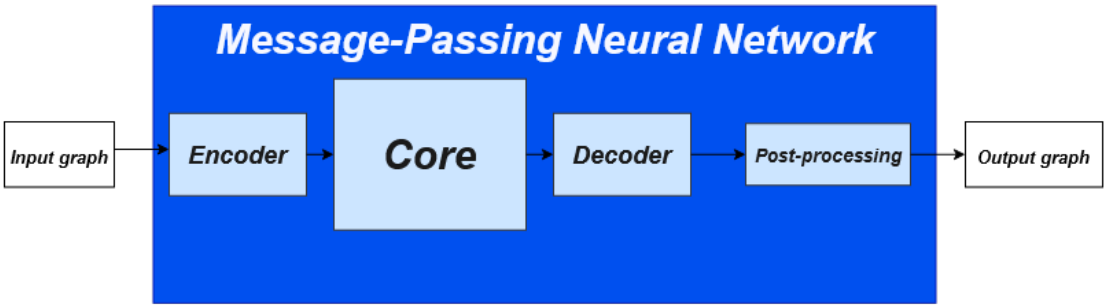


Figure 4.4: Internal architecture of the MPNN block.

The first three blocks have the following functionalities:

- *Encoder*: takes the input graph and encodes the nodes and edges attributes independently. This is possible due to two independent MLPs, the first encoding the nodes and the second encoding the edges. Both the MLPs present 2 fully-connected hidden layers with 32 neurons each (fig. 4.5), and return outputs in the form of 32-entries vectors.

- *Core*: takes as input the concatenation of the Encoder's output and the previous output of the Core, and performs the message-passing steps (Fig. 4.6). It is composed by three independent MLPs with 2 fully-connected hidden layers and 32 neurons each (Fig. 4.5), respectively responsible for edge features update, node features update and global feature update, according to the computational steps explained in Chapter 3.

- *Decoder*: decodes the edges and nodes attributes independently on each message-passing step, returning a labeled output graph. The Decoder is composed by two MLPs identical to the Encoder ones; the first is responsible for edges decoding and the second for nodes decoding.

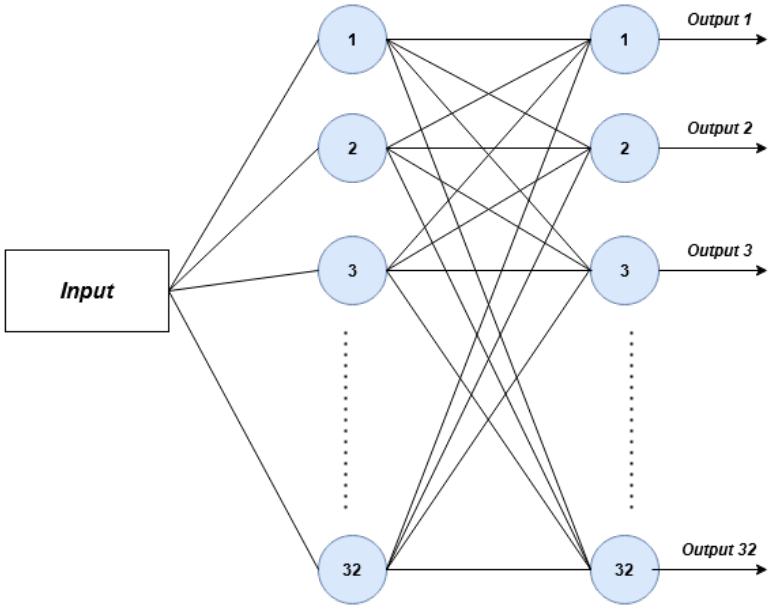


Figure 4.5: Fully connected MLP with 2 hidden layers, each consisting of 32 neurons.

Finally, after a prefixed number of message-passing steps, the decoder returns an output graph, which presents nodes and edges labeled by means of vectors of the same dimension of the target labels (2-entries vectors for nodes, $(W+1)$-entries vectors for edges). These labels represent the MPNN's solution to the input RWA problem: based on their labels, some nodes and edges will be considered as part of the solution and all the others as not part of the solution.
However, such output labels are not one-hot vectors as the target ones, but present entries which are real numbers. Therefore, a simple post-processing phase is applied to the decoder output in order to transform the output labels in one-hot vectors,

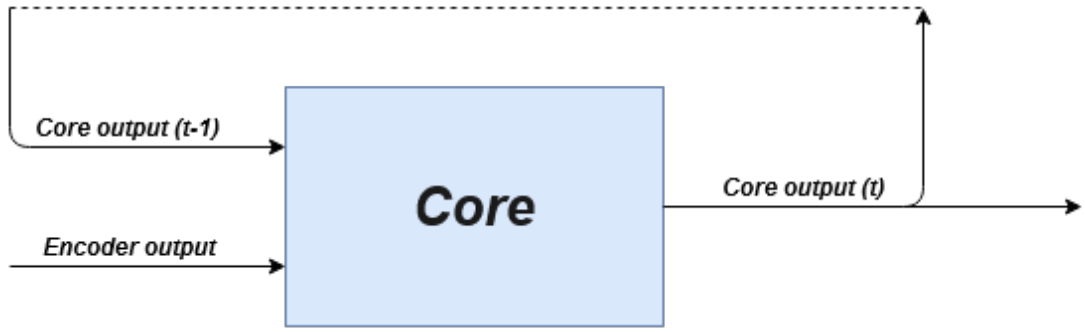


Figure 4.6: Core block.

and hence make the comparison with the target graph possible.
To explain such a post-processing computation, nodes labels are considered for simplicity, but the same concept holds for the edges labels. Given a node $v \in V$ and its output label $[v_1, v_2]$, with v_1 and v_2 real numbers, the index $class_v$ is computed as:

$$class_v = argmax([v_1, v_2]) \tag{4.1}$$

Therefore, $class_v$ will be equal to the index of the maximum value of v's output label. Since nodes labels have two entries, $class_v$ can either be 0 or 1. For edges, instead, such an index can assume values between 0 and W.
Next, v's resulting label will be a one-hot vector of 2 entries, in which the entry at index $class_v$ is set to 1. Considering the solution of Fig. 4.2, an example of MPNN's correct node labels prediction can be found in Table 4.6.

Node	Node label	$class_v$	Final output label	Solution
1	[5.0436,-2.3348]	0	[1,0]	No
2	[-4.1893,3.1344]	1	[0,1]	Yes
3	[1.4254,-1.1662]	0	[1,0]	No
4	[-2.1921,1.9847]	1	[0,1]	Yes
5	[-0.9093,0.4524]	1	[0,1]	Yes
6	[-4.6837,3.1179]	1	[0,1]	Yes

Table 4.6: Example of MPNN's node output labels for the solution of Fig 4.2.

In conclusion, by applying this procedure to every node and edge label, the final output graph will present nodes labels which are 2-entries one-hot vectors and edge labels which are $(W + 1)$-entries one-hot vectors. Using the same labels format of the target graphs, all the nodes with label [0,1] and all the edges with

label different from [1,0,..,0] are considered as part of the solution, which is the final answer of the MPNN when queried with the considered input graph.

4.2.3 Loss Computation and Optimization

At this point of the training iteration, there is an output graph, returned by the MPNN, and a target graph, obtained through the training algorithm. In order to understand how good is the quality of the MPNN's prediction, the output graph must be compared with the target graph. Such a comparison is called *loss computation*.
Within our framework, the loss computation is performed by means of a particular type of loss (or cost) function, which is the *Softmax Cross-Entropy* (SCE) function. The idea of such a loss function is that every input (nodes and edges in our case) can belong to one and only one class. Therefore, the SCE function returns the output values in the form of probability distribution.
In general, the SCE loss function takes as input a C-dimensional vector z and returns a C-dimensional vector y of real values between 0 and 1, representing the estimated probability distribution. In particular, every y_c, for $c = 1, ..., C$, is computed as an exponential normalization:

$$y_c = \frac{e^{z_c}}{\sum_{k=1}^{C} e^{z_k}} \qquad for \;\; c = 1, ..., C \tag{4.2}$$

The denominator $\sum_{k=1}^{C} e^{z_k}$ acts as a regularizer to make sure that $\sum_{c=1}^{C} y_c = 1$. More details about the SCE loss function can be found at [20].
After the loss computation, the resulting SCE must be optimized, in order to allow the MPNN to adjust its weights and hence return better and better predictions. Within our framework, the optimization process is performed by means of the TensorFlow's ADAM optimizer. Such an optimizer implements the ADAM optimization algorithm, which is an extension of the stochastic gradient descent. For more details about ADAM, the reader is invited to refer to [21].

4.2.4 Hyperparameters

The previous paragraphs have provided a thorough overview about the process of one single training iteration. Of course, such a process is repeated for the entire training phase duration, in the hope that the MPNN finally learns how to perform the assigned task.
However, given a topology on which the RWA problem is performed for multiple requests, and given our framework, the MPNN can be trained and hence learn how to solve the RWA problem in several different ways. In fact, a training phase is regulated by a set of *hyperparameters*. If the training data are always the same,

then any combination of hyperparameters leads to a unique and repeatable training phase for the MPNN.
All the hyperparameters that can be set-up within our framework are explained as follows.

Number of Iterations

The most intuitive hyperparameter is the number of iterations, n_{train}. It simply sets-up the total number of iterations that must be executed for the training phase. Within this framework, typical values of n_{train} are in the order of 10^4.

Batch Size

The batch size hyperparameter, $batch_{train}$, represents the number of input graphs which feed the MPNN at every iteration. This means that the total number of input graphs which are generated for the training phase is $n_{train}batch_{train}$. The first iteration uses the first $batch_{train}$ graphs, the second iteration uses the second $batch_{train}$ graphs, and so on. Notice that the total $n_{train}batch_{train}$ input graphs represent the sequence of calls during the time window, and hence every batch immediately follows the previous one along the timeline.
If $batch_{train} > 1$, it means that at every iteration the MPNN is fed with more than one input graph, and hence the loss computation is performed on the average comparison between input and corresponding target graphs.
Within this frameworks, typical values of $batch_{train}$ are in the order of 10^1-10^2.

Number of Messaging Steps

The number of messaging steps, n_{steps}, indicates the number of message-passing steps performed by the MPNN on every input graph. Within this framework, typical values of n_{steps} are between 6 and 20.

Learning Rate

The learning rate, l_{rate}, is the hyperparameter that regulates the speed of the optimization process. In fact, at the end of every training iteration, the loss function is moved towards its minimum. The value of l_{rate} represents the magnitude of the moving step.
The learning rate is always a trade-off between the learning time and the performance. Indeed, a small l_{rate} allows to perform short steps along the trajectory of the loss function, but the training will take more time. A large l_{rate}, instead, reduces significantly the training time, but due to larger steps there is the risk of

skipping some local minima of the function.
Within this framework, typical values of l_{rate} are between 10^{-4} and 10^{-1}.

4.3 Test Phase and Recovery Algorithm

4.3.1 Test Phase

Once the training phase is over, the MPNN is expected to have learned how to perform the assigned task, which is solving the RWA problem on the given network topology. In order to quantitatively measure the performance of the trained MPNN, a test phase is performed.
The test phase consists in generating a batch of N input graphs and give them as input to the MPNN. For any of these input graphs, the MPNN returns a prediction, i.e., an output graph. Differently from the training phase, where the output graph is compared to the corresponding target graph for the loss computation and optimization, here the comparison is done just to see if the MPNN's prediction to the input graph is either correct or incorrect. Due to the absence of optimization, the MPNN's weights are fixed, and hence, given an input graph, the corresponding output graph will be always the same.
For the law of large numbers, it has been proved that, if N is large enough, for any possible combination of input graphs the test results always converge to the same point. Therefore, such test results represent the absolute performance of the MPNN in solving the RWA problem on the given topology.
All the test performance metrics are defined at the end of this section. Moreover, our framework implements a recovery algorithm with the objective of recovering, if possible and by means of light weight operations, the outputs of queries which have not been solved correctly by the MPNN. Such an algorithm is discussed in detail in the following paragraph.

4.3.2 Recovery Algorithm

Let's consider one single input graph $graph_{in}$, picked from the test batch of N input graphs. When the MPNN test is performed on $graph_{in}$, two possible scenarios may arise (Fig. 4.7): either the MPNN has correctly solved $graph_{in}$ returning a correct output graph $graph_{out}$; or the MPNN has not correctly solved $graph_{in}$, returning a incorrect output graph $graph_{out}$. It is important to highlight that by incorrect $graph_{out}$ we mean an output graph which has been solved differently from the target one, but still it might be a feasible solution, as the reader will see through the algorithm description.

If $graph_{out}$ is correct, the MPNN has correctly solved the input RWA problem and there is nothing more to do. If $graph_{out}$ is incorrect, instead, it is given as input to the *recovery algorithm* (Fig 4.8).

The recovery algorithm is a post-processing algorithm implemented within our framework and has the objective of recovering, if possible, an incorrect output

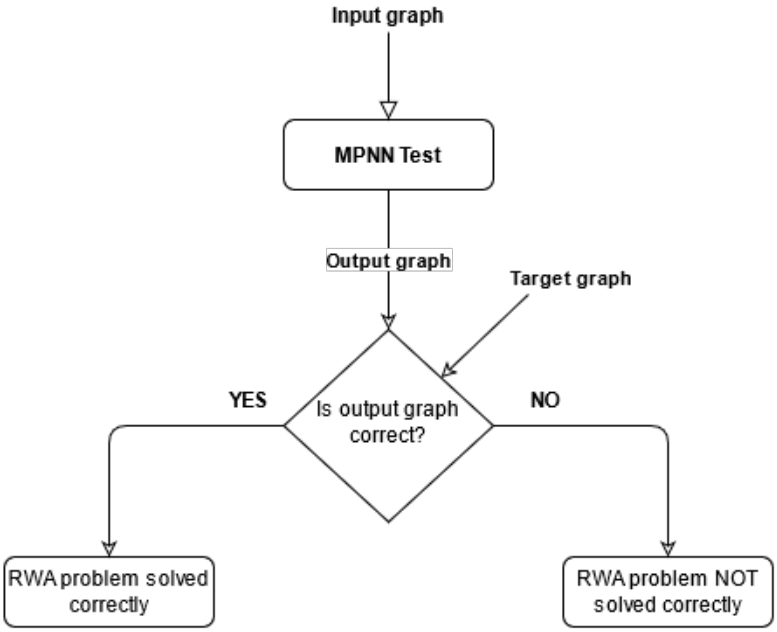


Figure 4.7: MPNN test: two possible scenarios.

graph by means of light weight operations. By *light weight operations*, we mean operations which are not computationally expensive. Therefore, at the end of the recovery algorithm, there are some chances that the incorrect $graph_{out}$ has been recovered, and hence has become correct. Applied to all the incorrect graphs of the test size, such a process significantly increases the test performances, as will be observed in the Chapter 5.
The recovery algorithm is straightforward and its flow chart can be observed in Fig. 4.9. Every step of the algorithm is discussed in detail in the following.

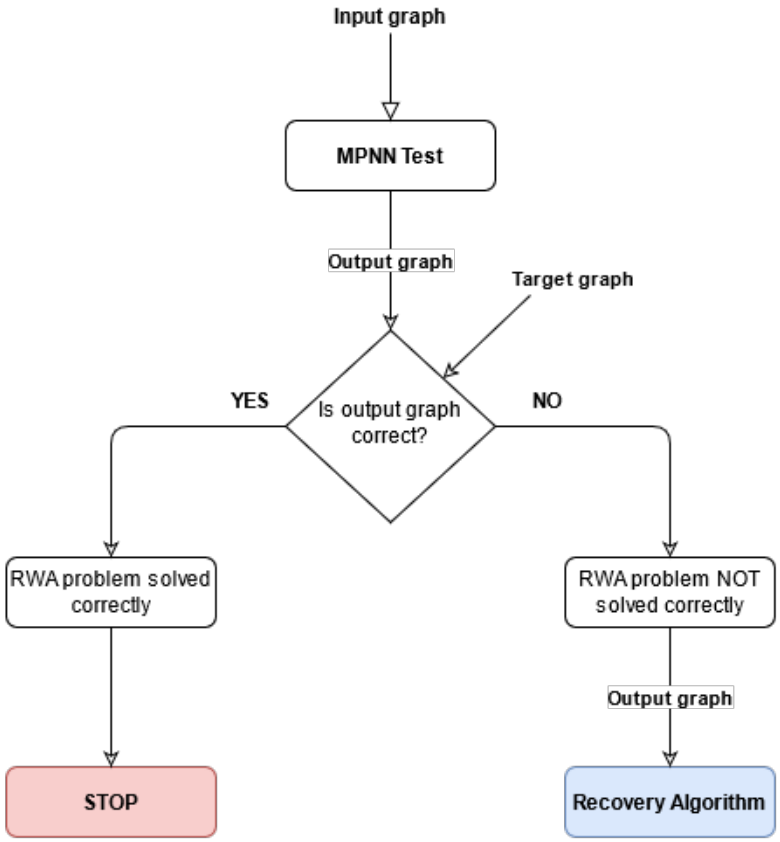


Figure 4.8: MPNN test: if $graph_{out}$ is incorrect, it is given as input to the recovery algorithm.

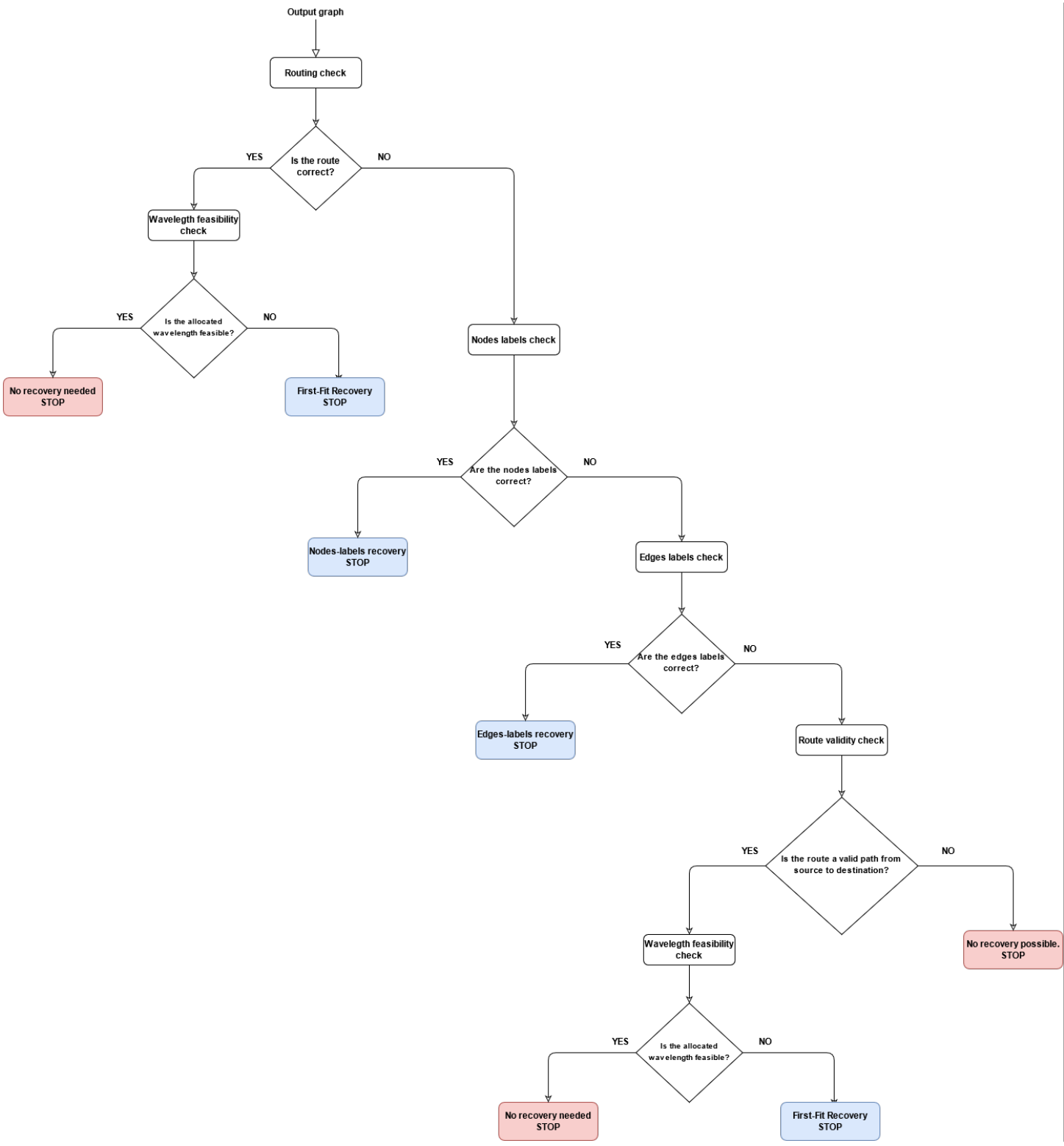


Figure 4.9: Recovery algorithm.

Step 1 - Routing Check

The first step of the recovery algorithm is to check whether the detected route is correct by considering nodes and edges labels, without looking at the allocated wavelength (Fig 4.10). If the route is correct, then also the wavelength allocation

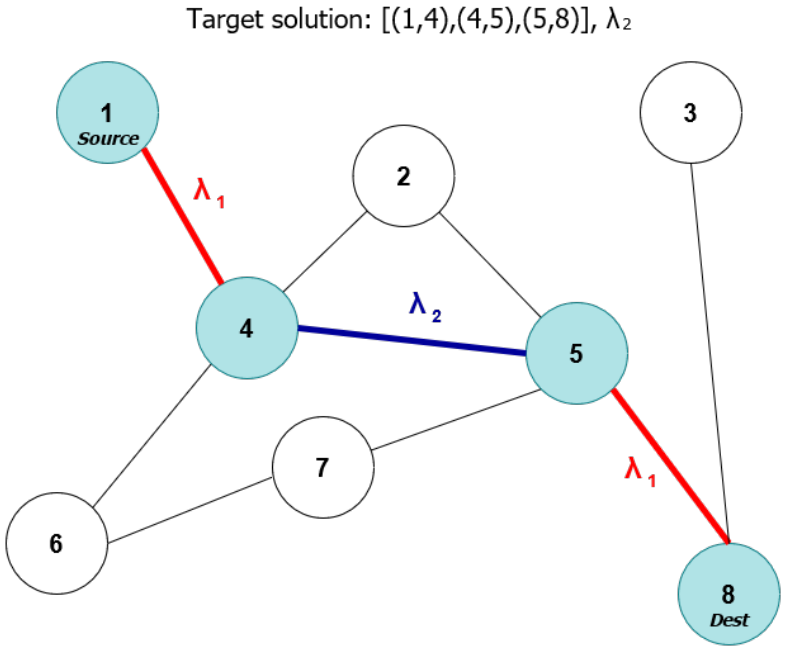


Figure 4.10: Example of routing check. Nodes labeled as part of the solution are cyan, whereas edges labeled as part of the solution are coloured according to the allocated wavelength.
Although the output solution is incorrect due to a wrong wavelength allocation, the detected route is the same of the target solution: [(1,4),(4,5),(5,8)]. Then, by applying a first-fit recovery, λ_2 is allocated.

on every edge of the route is considered. There are two possible scenarios:

1. the wavelength allocation is different from the target one, but still the allocated wavelength is feasible. In such a case, no recovery is needed, and $graph_{out}$ is automatically considered as recovered.

2. the wavelength allocation is different from the target one, and the allocated wavelength is unfeasible or two or more wavelengths have been allocated (note that such allocation is not possible due to the wavelength continuity constraint, see Chapter 2). In this case, the *first-fit recovery* is applied: the algorithm considers the set of edges of the route and allocates the first

available wavelength from the list of wavelengths, following a first-fit approach. Therefore, $graph_{out}$ is now considered as recovered.

Instead, if the detected route is not correct, the algorithm goes to Step 2.

Step 2 - Nodes Labels Recovery

The second step tries to recover $graph_{out}$ by only considering the nodes labels. In fact, since the route is incorrect, it must be that at least one among the set of nodes labels and the set of edges labels is incorrect. Therefore, this step leverages the nodes labels to recover all the graphs whose set of nodes labels is correct and set of edges labels is incorrect.
By considering the set of nodes labeled as part of the solution, if these are correct, it happens that starting from the source node and following the sequence of labeled nodes, the destination is reached. Hence, the target route is reconstructed by means of a *nodes labels recovery* (Fig. 4.11). At this point, only the edges which connect

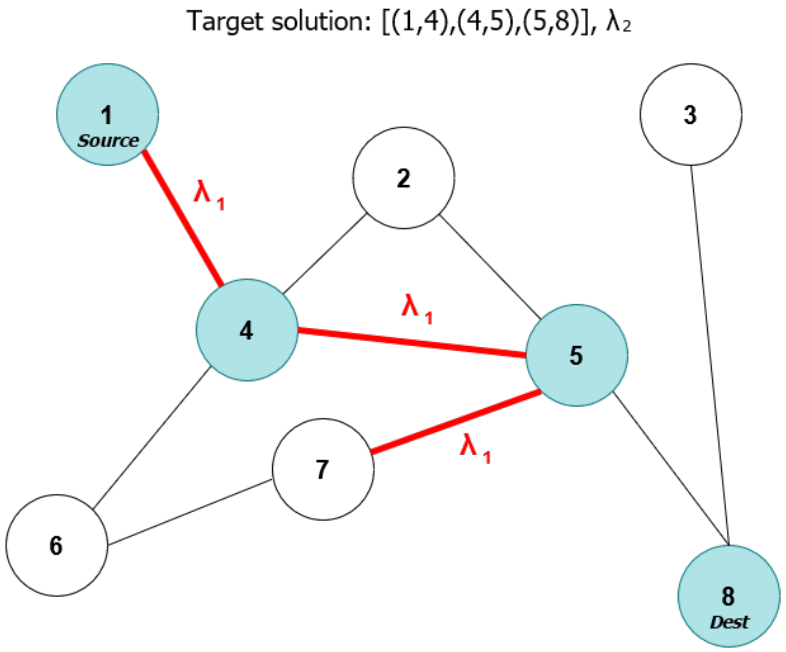


Figure 4.11: Example of nodes labels recovery. Nodes labeled as part of the solution are cyan, whereas edges labeled as part of the solution are coloured according to the allocated wavelength.
Although the output solution is incorrect, all the nodes labels are correct, and the target route [(1,4),(4,5),(5,8)] can be reconstructed. Then, by applying a first-fit recovery, λ_2 is allocated.

the route nodes are considered, and a first-fit algorithm is applied to allocate a proper wavelength. $graph_{out}$ is then considered as recovered.
Instead, if the nodes labels recovery does not work, the algorithm goes to Step 3.

Step 3 - Edges Labels Recovery

Step 3 tries to use an opposite approach to Step 2. In fact, since the nodes labels are for sure incorrect, *edges labels recovery* leverages the edges labels to recover all the graphs whose set of nodes labels is incorrect and set of edges labels is correct. By considering only the set of labeled edges, if these are correct, the target route is reconstructed by simply labeling as part of the solution only the nodes that are extremes of such edges (Fig. 4.12). However, it can also happen that the set of

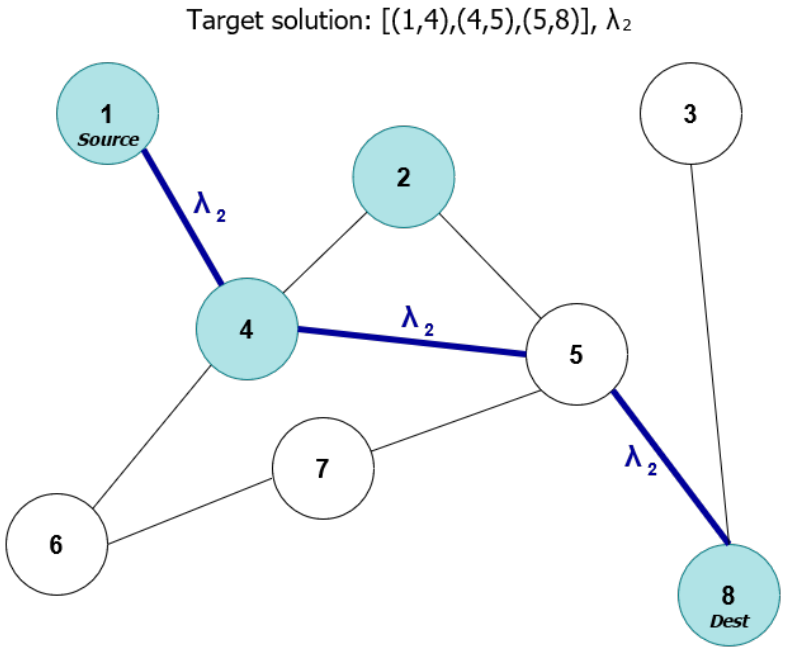


Figure 4.12: Example of edges labels recovery. Nodes labeled as part of the solution are cyan, whereas edges labeled as part of the solution are coloured according to the allocated wavelength.
Although the output solution is incorrect, all the edges labels are correct, and the target route [(1,4),(4,5),(5,8)] with allocated wavelength λ_2 can be reconstructed.

labeled edges form the target route with a wrong wavelength allocation. In this case, a first-fit recovery is applied and then, again, only nodes that are extremes of such edges are labeled as part of the solution (Fig. 4.13).

In both cases, $graph_{out}$ is considered as recovered. Instead, if the edges labels

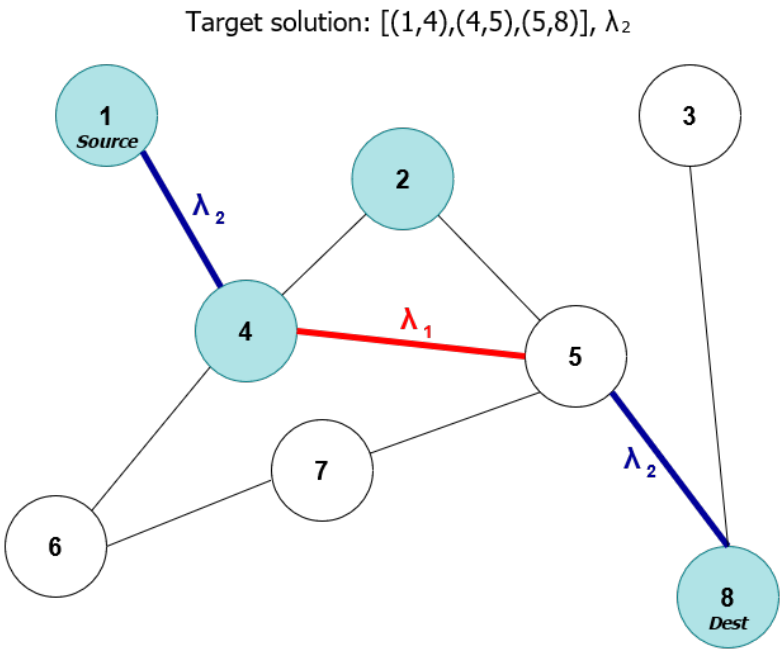


Figure 4.13: Example of edges labels recovery. Nodes labeled as part of the solution are cyan, whereas edges labeled as part of the solution are coloured according to the allocated wavelength.
Although the output solution is incorrect, all the edges labels are correct, and the target route [(1,4),(4,5),(5,8)] can be reconstructed. However, the wavelength allocation is incorrect, and, through a first-fit recovery, λ_2 is allocated.

recovery does not work, the algorithm goes to Step 4.

Step 4 - Route Validity Check

If the algorithm comes to this step, it means that both the set of nodes labels and the set of edges labels are incorrect. However, it may happen that the set of nodes labels and/or edges labels form a route different from the target one, but which is still a valid path from source to destination. Therefore, in order to check whether the reconstructed route is still a valid route, Step 4 leverages, in order, first both nodes and edges labels, then eventually only nodes labels, and finally eventually only edges labels (Fig. 4.14).
If the route is valid, two possible scenarios may arise:

1. the wavelength allocation is feasible, and $graph_{out}$ is automatically considered as recovered.

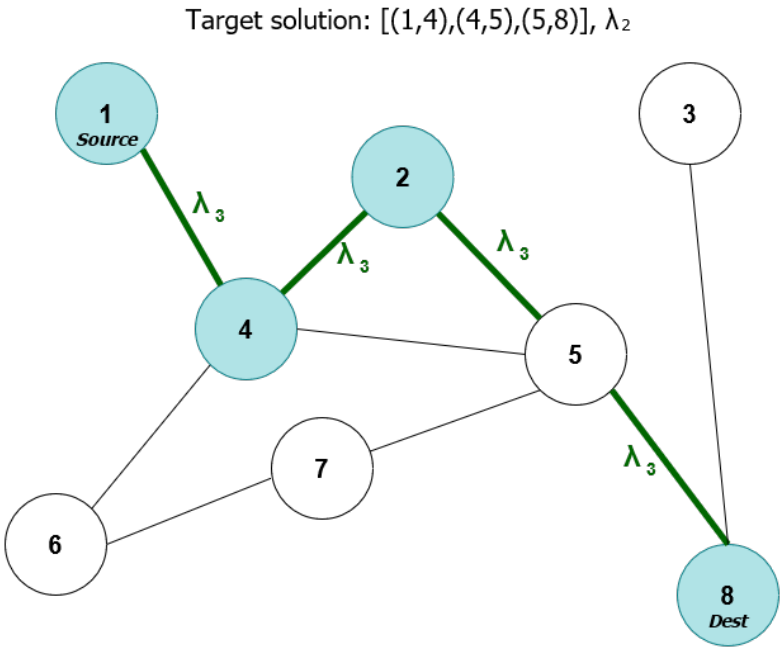


Figure 4.14: Example of route validity check. Nodes labeled as part of the solution are cyan, whereas edges labeled as part of the solution are coloured according to the allocated wavelength.
Although the output solution is incorrect, by following the edges labels, a valid route from source to destination is found: [(1,4),(4,2),(2,5),(5,8)]. Moreover, the wavelength allocation with λ_3 is feasible, and no first-fit recovery is needed.

2. the wavelength allocation is unfeasible or two or more wavelengths have been allocated. In this case, a first-fit recovery is applied and then $graph_{out}$ is considered as recovered.

If no valid route is found by Step 4 mechanisms, the recovery algorithm is terminated and $graph_{out}$ is definitely considered as not recoverable.

The whole procedure of the recovery algorithm is reported under form of pseudo-code in Algorithm 2.

Algorithm 2 Recovery algorithm

```
1: procedure RECOVERY(graph_out)
2:     check route only
3:     if route = correct then
4:         wavelength feasibility check
5:         if wavelength = feasible then
6:             graph_out is recovered
7:             STOP
8:         else
9:             first-fit recovery
10:            graph_out is recovered
11:            STOP
12:        end if
13:    else
14:        nodes labels check
15:        if nodes_labels = correct then
16:            nodes-labels recovery & first-fit recovery
17:            graph_out is recovered
18:            STOP
19:        else
20:            edges labels check
21:            if edges_labels = correct then
22:                edges-labels recovery
23:                wavelength feasibility check
24:                if wavelength = feasible then
25:                    graph_out is recovered
26:                    STOP
27:                else
28:                    first-fit recovery
29:                    graph_out is recovered
30:                    STOP
31:                end if
32:            else
33:                route validity check
34:                if route = valid then
35:                    if wavelength = feasible then
36:                        graph_out is recovered
37:                        STOP
38:                    else
39:                        first-fit recovery
40:                        graph_out is recovered
41:                        STOP
42:                    end if
43:                else
44:                    graph_out is not recoverable
45:                    STOP
46:                end if
47:            end if
48:        end if
49:    end if
50: end procedure
```

4.3.3 Performance Metrics

In the previous paragraphs, we have explained that a test must be performed in order to evaluate the MPNN's performance in solving the RWA problem. We have said that a test considers N input graphs, and that the MPNN returns an output graph for each of these input graphs. Next, in a post-processing phase, all the incorrect output graphs are subject to the recovery algorithm. Once the recovery phase is done, the test is terminated, and, by using the statistics collected during all these phases, the performance metrics for the MPNN can be computed.
First, let's define the following variables:

- N: total number of test graphs.
- $graphs_{RWA}$: number of output graphs correctly solved by the MPNN according to the RWA definition, and hence not subject to the recovery algorithm.
- $graphs_R$: number of incorrect output graphs whose route has been detected correctly by the routing check phase of the recovery algorithm.
- $graphs_{NOTrecoverable}$: number of incorrect output graphs that have not been recovered by the recovery algorithm.

The first performance metric is the *RWA accuracy*, which is defined as:

$$RWA_{accuracy} = \frac{graphs_{RWA}}{N} \tag{4.3}$$

and represents the percentage of test graphs correctly solved by the MPNN.
The second performance metric is the *routing accuracy*, which is defined as:

$$R_{accuracy} = \frac{graphs_{RWA} + graphs_R}{N} \tag{4.4}$$

and represents the percentage of test graphs whose route has been detected correctly by the MPNN.
The third performance metric is the *final accuracy*, which is defined as:

$$final_{accuracy} = \frac{N - graphs_{NOTrecoverable}}{N} \tag{4.5}$$

and represents the percentage of test graphs which have been solved correctly by the MPNN or have been recovered by the recovered algorithm.
Of course, it holds: $RWA_{accuracy} \leq R_{accuracy} \leq final_{accuracy}$. Moreover, it has been proved that, if N is large enough (i.e. the order of 10^4-10^5) all these accuracies converge to the same value for any test set, and hence represent the absolute performance of the MPNN in solving the RWA problem.

4.3.4 MPNN Implementation within a Network Scenario

Summarizing, once the MPNN is built and trained, it is tested to measure its performances in solving the RWA problem. If the final performances have not reached a satisfactory level, the MPNN can be re-trained in a different way in order to increase such final performances. Else, in case the achieved performances are satisfactory, the MPNN can be implemented within a realistic network scenario.
In fact the MPNN can be implemented within the network on which has been trained, and used on-demand as on-line solving tool to compute routes and allocate wavelengths for any incoming call request. Notice that the training phase and the test phase are performed off-line, and hence the MPNN can be provided by input graphs and corresponding target graphs during these phases. In a realistic implementation, instead, the MPNN works in an on-line fashion, which means the incoming call request must be allocated immediately, and no target graph is provided to compare the MPNN's output.
Therefore, when used on-line, the MPNN takes as input an incoming connection request and returns a decision for such request. Since the correctness of the decision is unknown due to the absence of target solution, the only thing that can be checked about the decision is its feasibility. In fact, it makes sense that the MPNN's decision is actually implemented if and only if such decision is feasible.
The feasibility of any MPNN's decision is established by the *feasibility algorithm*. This is a fast post-processing algorithm which works by leveraging some of the mechanisms used by the recovery algorithm during the test phase. When the feasibility algorithm is applied to a MPNN's decision, two possible scenarios may arise (Fig. 4.15):

1. the MPNN's decision is feasible, and can be implemented in the network, i.e., a route and a wavelength can actually be allocated for the incoming call, if this is accepted.

2. the MPNN's decision is unfeasible, and cannot be implemented in the network. In such a case, the decision is discarded and the network must rely on alternative tools able to take a feasible decision for the incoming call request.

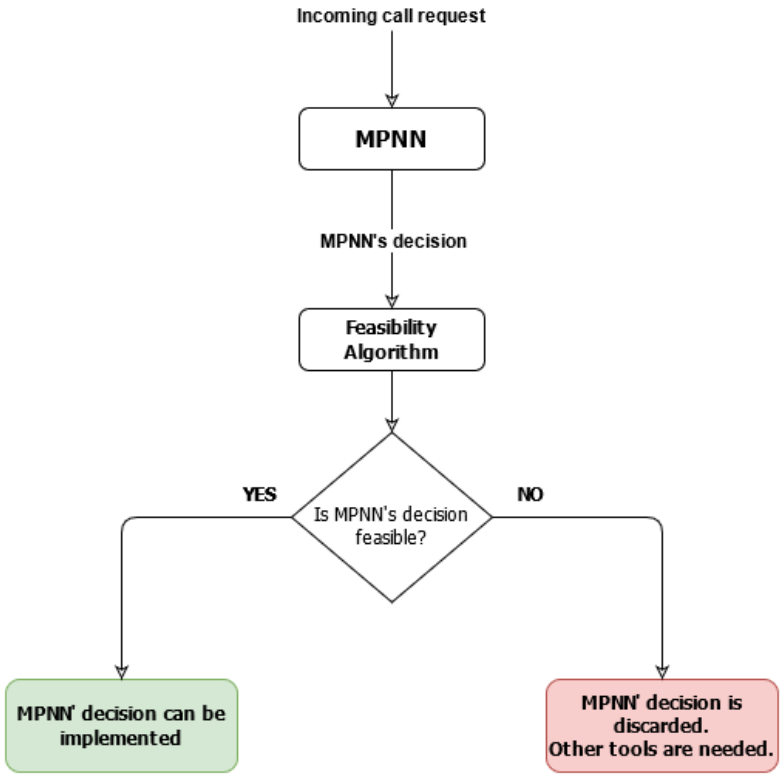


Figure 4.15: MPNN implementation within a network scenario. Any MPNN's decision is subject to the feasibility algorithm, which establishes whether such decision is either feasible or unfeasible.

As said above, the feasibility algorithm relies on the same mechanisms of the recovery algorithm and its flow chart can be observed in Fig. 4.16.
In the following, every step is briefly discussed.

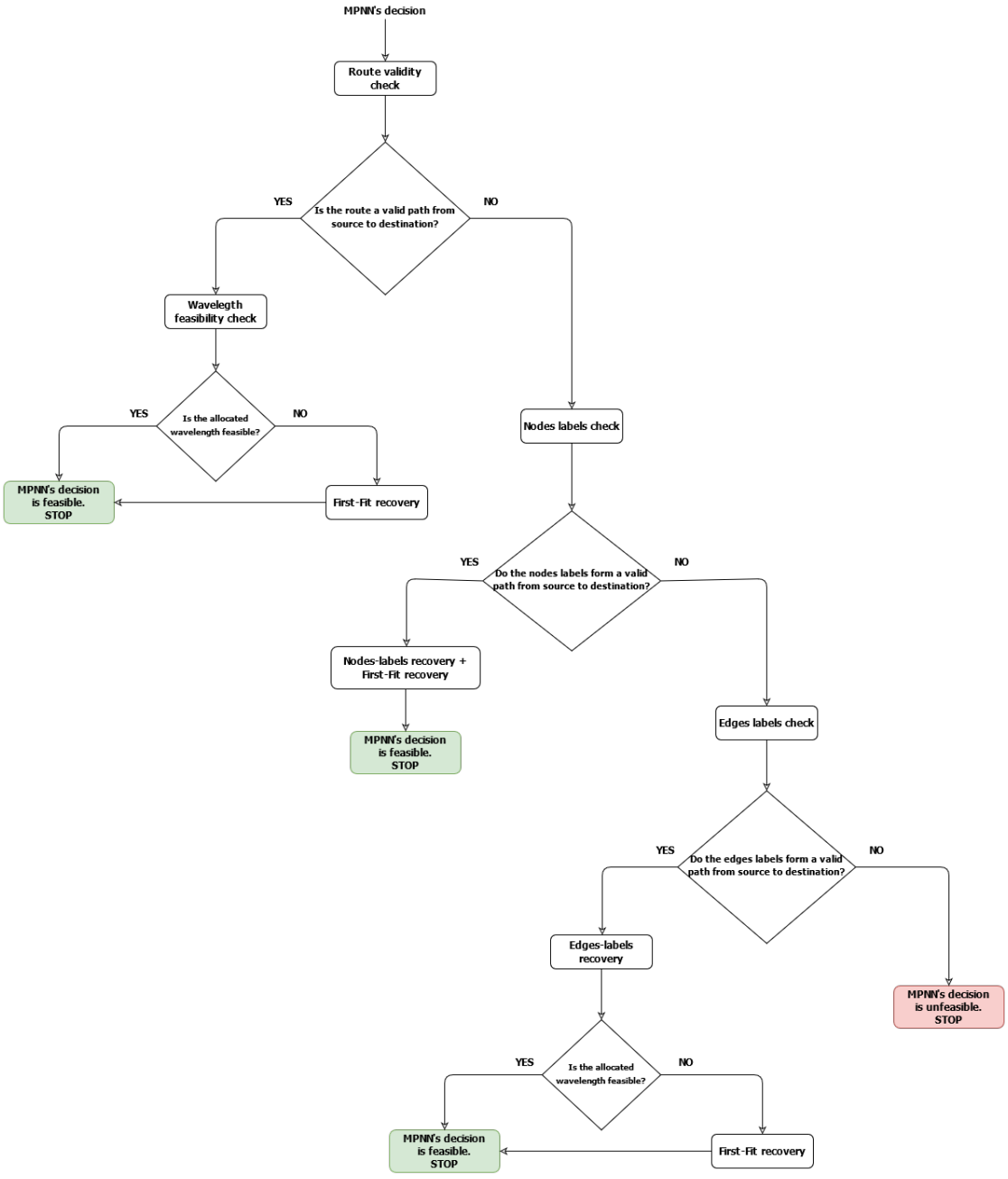


Figure 4.16: Feasibility algorithm.

Step 1 - Route Validity Check

The first step of the feasibility algorithm is to check whether the route is a valid path from source to destination. If the route is valid, then also the wavelength assignment is considered:

- If the wavelength assignment is feasible, then the MPNN's decision is considered feasible, and the algorithm is stopped.
- If the wavelength assignment is unfeasible, then a first-fit recovery is needed. After that, the MPNN's decision is considered feasible, and the algorithm is stopped.

Notice that there is no way to establish whether the valid route is the optimal one, due to the absence of target solution.
Instead, if the route is not a valid path from source to destination, the algorithm goes to Step 2.

Step 2 - Nodes Labels Recovery

This step considers only the nodes labels. If nodes labeled as part of the solution form a valid path from source to destination, then a nodes labels recovery is performed, meaning that the new route is built by labeling all the edges which connect the solution nodes. Next, a first-fit recovery is applied on such edges in order to allocate a feasible wavelength. The MPNN's decision is then considered feasible, and the algorithm is stopped.
Instead, if the nodes labels do not form a valid path from source to destination, the algorithm goes to Step 3.

Step 3 - Edges Labels Recovery

This step follows an opposite approach compared to the previous one, and considers only the edges labeled as part of the solution. If these form a valid path from source to destination, then a edges labels recovery is performed, meaning that the new route is built by labeling all the nodes which are extremes of the solution edges. Next, the wavelength assignment is considered:

- If the wavelength assignment is feasible, then the MPNN's decision is considered feasible, and the algorithm is stopped.
- If the wavelength assignment is unfeasible, then a first-fit recovery is needed. After that, the MPNN's decision is considered feasible, and the algorithm is stopped.

Instead, if the edges labels do not form a valid path from source to destination, the MPNN's decision is definitely considered unfeasible, and the algorithm is stopped. In such a case, the network must rely on different approaches from the MPNN to take a decision on the incoming call request.

The whole procedure of the feasibility algorithm is reported under form of pseudo-code in Algorithm 3.

Algorithm 2 Feasibility algorithm

```
1: procedure FEASIBILITY(decision_MPNN)
2:     route validity check
3:     if route = valid then
4:         wavelength feasibility check
5:         if wavelength = feasible then
6:             decision_MPNN is feasible
7:             STOP
8:         else
9:             first-fit recovery
10:            decision_MPNN is feasible
11:            STOP
12:        end if
13:    else
14:        route validity check through nodes labels
15:        if route_nodes = valid then
16:            nodes-labels recovery
17:            first-fit recovery
18:            decision_MPNN is feasible
19:            STOP
20:        else
21:            route validity check through edges labels
22:            if route_edges = valid then
23:                edges-labels recovery
24:                wavelength feasibility check
25:                if wavelength = feasible then
26:                    decision_MPNN is feasible
27:                    STOP
28:                else
29:                    first-fit recovery
30:                    decision_MPNN is feasible
31:                    STOP
32:                end if
33:            else
34:                decision_MPNN is unfeasible
35:                STOP
36:            end if
37:        end if
38:    end if
39: end procedure
```

4.4 Computational Complexity

This chapter has thoroughly presented how our GNN framework for the RWA problem works. First, a GNN with a particular architecture is built, which is the MPNN. This is trained, tested, and eventually implemented within a realistic network scenario to deal with the on-line connection requests. The purpose of this section is to analyze the computational complexity of such a powerful framework. The computational complexity is first computed separately for each fundamental block of the framework, and then, by merging the blocks together, the final complexity is obtained. Let's also remind the following notation from the previous sections:

- $|V|$: number of nodes in the network topology.
- $|E|$: number of edges in the network topology.
- $|W|$: number of initially available wavelengths on every edge in the network topology.

MPNN Complexity

In [22], it has been proved that the MPNN complexity for the shortest path problem is $O(|V|+|E|)$. Since in our framework every edge is encoded with two attributes, a real value for the distance and a $(W+1)$-entries vector for the available wavelengths, the MPNN complexity becomes:

$$C_{MPNN} = O(|V| + |W+1||E|) = O(|V| + |W||E|) \tag{4.6}$$

Recovery Algorithm Complexity

The complexity of the recovery algorithm is equal to the worst among the worst-case complexities for every possible final branch of the algorithm (refer to Fig 4.9).
Step 1 checks whether the route is correct ($O(|V| + |E|)$) and, if it is, performs a wavelength feasibility check and eventually a first-fit recovery ($O(|W||E|)$), and the algorithm terminates with complexity:

$$C_1 = O(|V|+|E|)+O(|W||E|) = O(|V|+|E|+|W||E|) = O(|V|+|W||E|) \tag{4.7}$$

If the route is incorrect, a nodes-labels check ($O(|V|)$) is performed; if the node labels are correct, a nodes-labels recovery ($O(|W||E|)$) is performed, and the algorithm terminates with complexity:

$$C_2 = O(|V| + |E|) + O(|V|) + O(|W||E|) = O(|V| + |W||E|) \tag{4.8}$$

If the node labels are incorrect, an edges labels check ($O(|E|)$) is performed; if the edges labels are correct, an edges labels recovery with wavelength feasibility check is performed ($O(|V|+|W|||E|)$), and the algorithm terminates with complexity:

$$C_3 = O(|V|+|E|)+O(|V|)+O(|E|)+O(|V|+|W|||E|) = O(|V|+|W||E|) \quad (4.9)$$

If the edges labels are incorrect, a route validity check is performed ($O(|V|+|E|)$); if the route is a valid path from source to destination, a wavelength feasibility check and eventually first-fit recovery are performed ($O(|W||E|)$), and the algorithm terminates with complexity:

$$C_4 = O(|V|+|E|)+O(|V|)+O(|E|)+O(|V|+|E|)+O(|W||E|) = O(|V|+|W||E|) \quad (4.10)$$

Finally, if the route is not valid, the algorithm terminates with complexity:

$$C_5 = O(|V|+|E|)+O(|V|)+O(|E|)+O(|V|+|E|) = O(|V|+|E|) \quad (4.11)$$

The worst-case complexities are hence C_1, C_2, C_3, C_4; notice that the best-case complexity is C_5 which represents the case of unrecoverable graph. Therefore, the recovery algorithm complexity is:

$$C_{recovery} = O(|V|+|W||E|) \quad (4.12)$$

Feasibility Algorithm Complexity

By simply following the same procedure of the recovery algorithm, the complexity of the feasibility algorithm is:

$$C_{feasibility} = O(|V|+|W||E|) \quad (4.13)$$

Final Complexity

By combining the previously computed complexities, we can conclude that the test complexity is:

$$C_{test} = C_{MPNN}+C_{recovery} = O(|V|+|W||E|)+O(|V|+|W||E|) = O(|V|+|W||E|) \quad (4.14)$$

whereas the MPNN implemented in a network scenario has complexity:

$$C_{implementation} = C_{MPNN}+C_{feasibility} = O(|V|+|W||E|)+O(|V|+|W||E|) = O(|V|+|W||E|)| \quad (4.15)$$

Therefore, any usage of the MPNN for the RWA problem has complexity O(|V|+|W||E|).

Chapter 5

Results

Having introduced and described in detail, in Chapter 4, the GNN framework for the RWA problem, it is finally time to report the numerical results that are achievable by such framework.
Four different experiments have been performed, each one considering a specific network scenario. Hence, this Chapter reports the numerical results regarding MPNN's performances in terms of achieved accuracies for each experiment, and an additional analysis regarding the time complexity in terms of execution times.
All the experiments have been performed by relying on the computational resources provided by the High Performance Computing (HPC) cluster [23], sited in Politecnico di Torino.

5.1 Performance Analysis: 1^{st} Experiment

5.1.1 Network Scenario

The 1^{st} experiment has been run on a medium-large-size network, which is the Interroute network (Fig. 5.1), whose structure has been provided by [24].

Figure 5.1: Interroute Network [24].

The network is connected and composed by 96 nodes, and all its features are reported in Table 5.1.

Network feature	Value
number of nodes	96
number of edges	119
density	0.03
average degree	2.47
average distance	7.91
diameter	21
clustering	0.08

Table 5.1: Features of the Interroute network.

In this experiment, every link of the network has been initially allocated 80 available wavelengths. In order to train and test the MPNN within a network traffic scenario with negligible blocking probability, a low-traffic load must be set-up within the network.
To find such a proper set-up, a simulation of call requests with specific values for the traffic regulator parameters, i.e. λ and μ, has been run a-priori. All the requests

are solved accordingly to the fixed-routing approach implemented with Dijkstra's algorithm, followed by the First-Fit WA, and the final blocking probability is computed. If the blocking probability is not the desired one, the simulation is re-run until a satisfactory set of values for λ and μ is found.
In particular, for the Interroute network with 80 available wavelengths it has been chosen to set λ and μ in such a way that $\lambda\mu = 155$, which corresponds to a network load of 13.49% and a blocking probability equal to $1.35 \cdot 10^{-3}$ in the long-run.

5.1.2 Analysis Set-Up

Given the above defined network scenario, the MPNN must be trained and finally tested in order to obtain a quantitative measure of how confident is the MPNN in solving the RWA problem in such a scenario.
The training algorithm is the fixed-routing approach implemented with Dijkstra's algorithm, followed by the First-Fit WA. Since the training phase is regulated by the set-up of the hyperparameters described in Chapter 4, the analysis has been performed over the variation of such hyperparameters.
In particular, the following default hyperparameters have been chosen:

- $n_{train} = 20.000$
- $batch_{train} = 75$
- $n_{steps} = 18$
- $l_{rate} = 10^{-2}$

Therefore, every analysis on a specific hyperparameter has been performed by varying such hyperparameter and by keeping all the others fixed to the default values. Then, after the training phase, the MPNN is tested over a batch of 100.000 call requests, fixed for any analysis. Moreover, every analysis has been run 10 times with different seeds, and all the presented results have been obtained by averaging.

5.1.3 Numerical Results

Iterations Analysis

The analysis on the number of iterations has been performed by varying n_{train} over the values: 10.000, 20.000, 30.000, 40.000. The results in terms of test accuracies are resumed in Table 5.2, and can be graphically observed in Fig. 5.2(A).

The first observation one can have is that all the accuracies for the RWA problem are extremely low. This is due to the large amount of available wavelengths, and hence to the fact that the First-Fit approach initially teaches the MPNN to use

Accuracy	$n_{train} = 10.000$	$n_{train} = 20.000$	$n_{train} = 30.000$	$n_{train} = 40.000$
$\text{RWA}_{accuracy}$	3.74%	4.87%	5.12%	5.05%
$\text{R}_{accuracy}$	32.78%	66.51%	67.67%	78.97%
$\text{final}_{accuracy}$	68.69%	91.47%	94.83%	94.84%

Table 5.2: Iterations analysis - results for the Interroute network with 80 wavelengths. The confidence interval for the accuracies is negligible.

only the first wavelengths in the list.
For what concerns the other metrics, instead, it is clear that 10.000 iterations are not enough for a satisfactory MPNN's learning. In particular, the routing accuracy increases as the number of iterations increases, but all the cases $n_{train} = 20.000, 30.000, 40.000$ similarly achieve an excellent final accuracy. In conclusion, 30.000 iterations should represent a good compromise between performance and training time (see Section 5.5.1).
In general, these results are the first evidence that the recovery algorithm is able to drastically improve the MPNN's performances, and hence represents a fundamental post-processing phase for the framework.

Messaging Steps Analysis

The analysis on the number of messaging steps has been performed by varying n_{steps} over the values: 14, 16, 18, 20, 22. The results in terms of test accuracies are resumed in Table 5.3, and can be graphically observed in Fig. 5.2(B).

Accuracy	$n_{steps} = 14$	$n_{steps} = 16$	$n_{steps} = 18$	$n_{steps} = 20$	$n_{steps} = 22$
$\text{RWA}_{accuracy}$	5.38%	4.86%	4.87%	5.06%	3.60%
$\text{R}_{accuracy}$	70.58%	68.16%	66.51%	26.59%	30.79%
$\text{final}_{accuracy}$	93.89%	91.61%	91.47%	48.43%	58.00%

Table 5.3: Messaging steps analysis - results for the Interroute network with 80 wavelengths. The confidence interval for the accuracies is negligible.

By looking at the results, it looks clear that performing 14 rounds of messaging steps is enough for achieving excellent performances for what concerns routing and final accuracies, but the cases $n_{steps} = 16,18$ achieve satisfactory results as well. Instead, the cases $n_{steps} = 20,22$ see their performances significantly drop, probably due to an overfitting effect given by the massive number of messaging steps.

Learning Rate Analysis

The analysis on the learning rate has been performed by varying l_{rate} over the values: 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}. The results in terms of test accuracies are resumed in Table 5.4, and can be graphically observed in Fig. 5.2(C).

Accuracy	$l_{rate} = 10^{-1}$	$l_{rate} = 10^{-2}$	$l_{rate} = 10^{-3}$	$l_{rate} = 10^{-4}$
$\text{RWA}_{accuracy}$	1.29%	4.87%	5.10%	1.15%
$\text{R}_{accuracy}$	10.65%	66.51%	61.42%	3.58%
$\text{final}_{accuracy}$	20.46%	91.47%	90.73%	18.02%

Table 5.4: Learning rate analysis - results for the Interroute network with 80 wavelengths. The confidence interval for the accuracies is negligible.

The results clearly show that the learning rate trade-off is achieved for values between 10^{-3} and 10^{-2}, corresponding to excellent routing and final accuracies. The other two cases, instead, lead to poor performances: the case 10^{-4} probably due to a too small rate, and hence a not sufficient amount of learning time; the case 10^{-1} probably due to a too large rate and hence the skip of some local minima of the loss function.

Batch Size Analysis

The analysis on the batch size has been performed by varying $batch_{train}$ over the values: 25, 50, 75, 100. The results in terms of test accuracies are resumed in Table 5.5, and can be graphically observed in Fig. 5.2(D).

Accuracy	$batch_{train} = 25$	$batch_{train} = 50$	$batch_{train} = 75$	$batch_{train} = 100$
$\text{RWA}_{accuracy}$	4.24%	4.25%	4.87%	4.31%
$\text{R}_{accuracy}$	37.81%	40.12%	66.51%	62.98%
$\text{final}_{accuracy}$	78.47%	75.78%	91.47%	77.87%

Table 5.5: Batch size analysis - results for the Interroute network with 80 wavelengths. The confidence interval for the accuracies is negligible.

The results demonstrate that every value of batch size is able to achieve quite good performances, but the case $batch_{train} = 75$ represent the best compromise and achieves excellent results. The cases $batch_{train} = 25, 50$ do not probably include a large enough number of training data in every iteration, whereas the case $batch_{train} = 100$ leads to a too large batch and hence to a worse ability for the MPNN to adjust its weights.

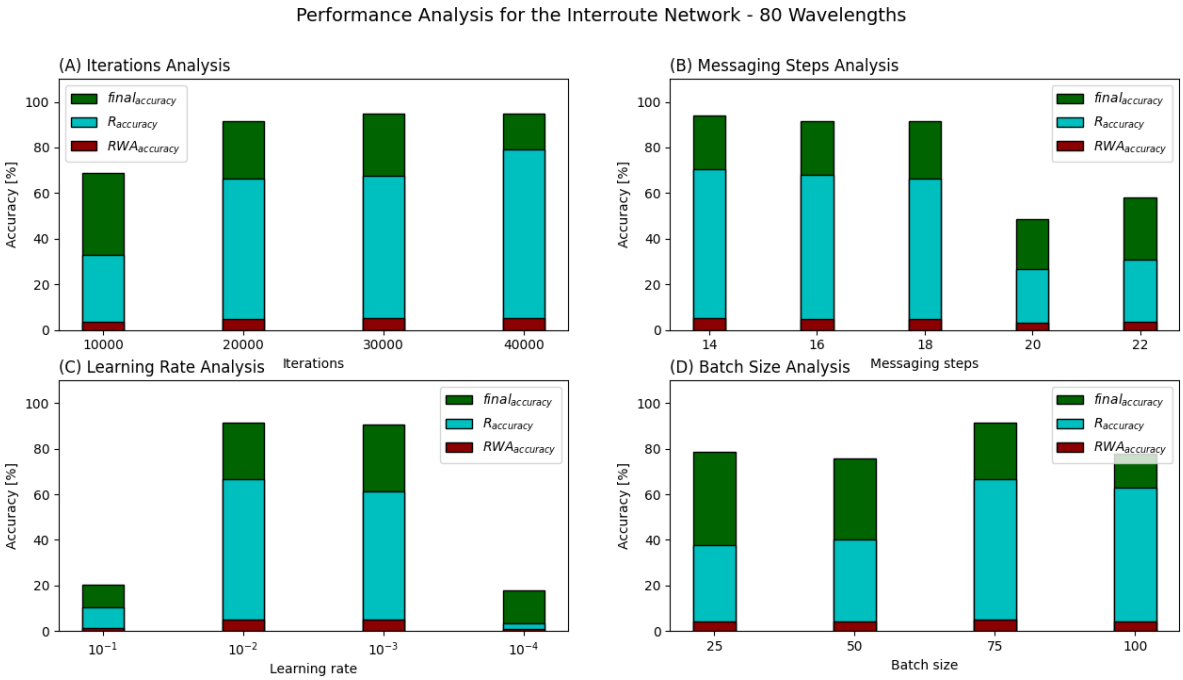


Figure 5.2: Graphical performance results for the Interroute network with 80 wavelengths.

5.2 Performance Analysis: 2^{nd} Experiment

5.2.1 Network Scenario and Analysis Set-Up

The 1^{st} experiment has proven that, in a network with a large number of available wavelengths, the MPNN is not able to learn how to solve the RWA problem without relying on the recovery algorithm. Therefore, the 2^{nd} experiment aims at observing whether the RWA accuracy can increase by simply reducing wavelengths space.
The network scenario is still composed by the Interroute network, with the WA sub-problem reduced to the extreme case of 1 available wavelength on every link. In order to keep the blocking probability negligible, the traffic arrival process has been adapted to obtain a similar network load as the previous experiment. Since the available resources have been drastically reduced, the traffic load has been set up with parameters $\lambda\mu = \frac{1}{200}$, which corresponds to a network load of 0.03% and a blocking probability equal to $1.73 \cdot 10^{-3}$ in the long-run.
The same analysis of the 1^{st} experiment with identical methods has been repeated.

5.2.2 Numerical Results

Iterations Analysis

The analysis on the number of iterations has been again performed by varying n_{train} over the values: 10.000, 20.000, 30.000, 40.000. The results in terms of test accuracies are resumed in Table 5.6, and can be graphically observed in Fig. 5.3(A).

Accuracy	$n_{train} = 10.000$	$n_{train} = 20.000$	$n_{train} = 30.000$	$n_{train} = 40.000$
$\text{RWA}_{accuracy}$	13.20%	32.73%	45.03%	72.18%
$\text{R}_{accuracy}$	14.41%	33.84%	45.78%	72.71%
$\text{final}_{accuracy}$	26.91%	49.58%	62.20%	82.04%

Table 5.6: Iterations analysis - results for the Interroute network with 1 wavelength. The confidence interval for the accuracies is negligible.

These first results immediately show that the drastic reduction of the wavelengths space has lead to a significant improvement of the RWA accuracy, even achieving excellent values with 40.000 iterations.
In case of 1 available wavelength, the routing accuracy is not exactly equal to the RWA one, but is always slightly better. This is due to the fact that, in case of blocked call, the RWA problem is considered corrected if and only if the output is a blocked call, whereas the routing sub-problem is considered correct if the output route is anyway correct. Finally, one can observe that the recovery algorithm is not able to achieve the same excellent performances of the 1^{st} experiment, probably due to the fact that with only one available wavelength there is no first-fit recovery to apply.

Messaging Steps Analysis

The analysis on the number of messaging steps has been again performed by varying n_{steps} over the values: 14, 16, 18, 20, 22. The results in terms of test accuracies are resumed in Table 5.7, and can be graphically observed in Fig. 5.3(B).

Accuracy	$n_{steps} = 14$	$n_{steps} = 16$	$n_{steps} = 18$	$n_{steps} = 20$	$n_{steps} = 22$
$\text{RWA}_{accuracy}$	70.00%	34.21%	32.73%	36.05%	4.91%
$\text{R}_{accuracy}$	70.51%	35.38%	33.84%	36.63%	5.20%
$\text{final}_{accuracy}$	86.71%	58.89%	49.58%	55.37%	16.96%

Table 5.7: Messaging steps analysis - results for the Interroute network with 1 wavelength. The confidence interval for the accuracies is negligible.

Again, the RWA accuracies have increased significantly with respect to the 1^{st} experiment. The best performances are sharply achieved by the case $n_{steps} = 14$, meaning that with only one wavelength less rounds of messaging steps are needed. In fact, all the other cases do not achieve good results, and the case $n_{steps} = 22$ is even extremely unsatisfactory.

Learning Rate Analysis

The analysis on the learning rate has been again performed by varying l_{rate} over the values: 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}. The results in terms of test accuracies are resumed in Table 5.8, and can be graphically observed in Fig. 5.3(C).

Accuracy	$l_{rate} = 10^{-1}$	$l_{rate} = 10^{-2}$	$l_{rate} = 10^{-3}$	$l_{rate} = 10^{-4}$
$\text{RWA}_{accuracy}$	4.49%	32.73%	58.69%	3.05%
$\text{R}_{accuracy}$	4.81%	33.84%	68.60%	5.34%
$\text{final}_{accuracy}$	9.74%	49.58%	82.14%	13.61%

Table 5.8: Learning rate analysis - results for the Interroute network with 1 wavelength. The confidence interval for the accuracies is negligible.

As for the previous experiment, the extreme cases $l_{rate} = 10^{-1}$ and $l_{rate} = 10^{-4}$ are completely inappropriate. The other two cases achieve a good RWA accuracy, but this time the case $l_{rate} = 10^{-3}$ is significantly better than $l_{rate} = 10^{-2}$, meaning that with only one wavelength a smaller rate is more suited.

Batch Size Analysis

The analysis on the batch size has been again performed by varying $batch_{train}$ over the values: 25, 50, 75, 100. The results in terms of test accuracies are resumed in Table 5.9, and can be graphically observed in Fig. 5.3(D).

Accuracy	$batch_{train} = 25$	$batch_{train} = 50$	$batch_{train} = 75$	$batch_{train} = 100$
$\text{RWA}_{accuracy}$	11.56%	16.46%	32.73%	31.16%
$\text{R}_{accuracy}$	12.32%	17.02%	33.84%	32.37%
$\text{final}_{accuracy}$	29.04%	36.80%	49.58%	57.66%

Table 5.9: Batch size analysis - results for the Interroute network with 1 wavelength. The confidence interval for the accuracies is negligible.

Again, it seems that the cases $batch_{train} = 25, 50$ do not include enough training data to achieve satisfactory results. The other two cases are more acceptable, but

lead to significantly worse performances with respect to the case of 80 wavelengths, highlighting again the worse effectiveness of the recovery algorithm.

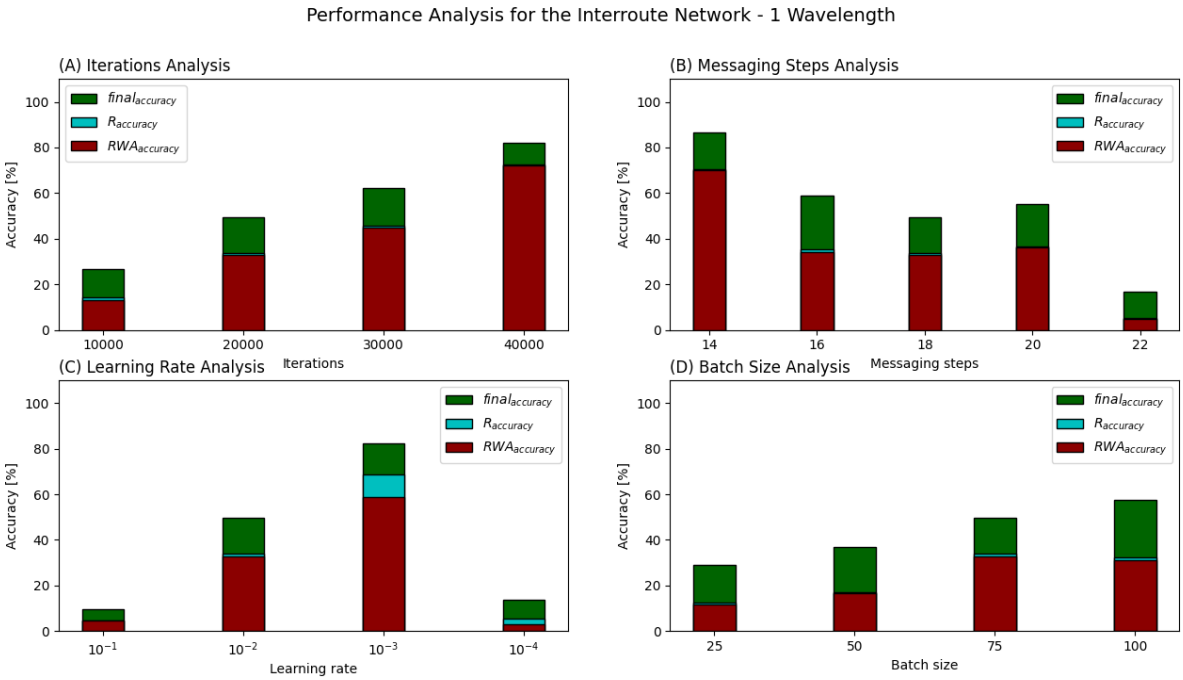


Figure 5.3: Graphical performance results for the Interroute network with 1 wavelength.

5.3 Performance Analysis: 3^{rd} Experiment

5.3.1 Network Scenario and Analysis Set-Up

The previous sections have reported the numerical results of the MPNN's performances in solving the RWA problem in a medium-large-size network. It turns out that, for some specific hyperparameters set-up, the achieved performance in terms of accuracies can reach very high values.
Now, the 3^{rd} experiment aims at testing the MPNN over the same hyperparameters range but within a smaller network scenario, in order to see whether the performances can even increase by reducing the network size. Therefore, this experiment has been run on a small-size network, which is the UUNET network (Fig. 5.4), whose structure has been provided by [24].

The features of such a network have been computed and reported in Table 5.10. In particular, one can notice that such a network has a nodes cardinality which is

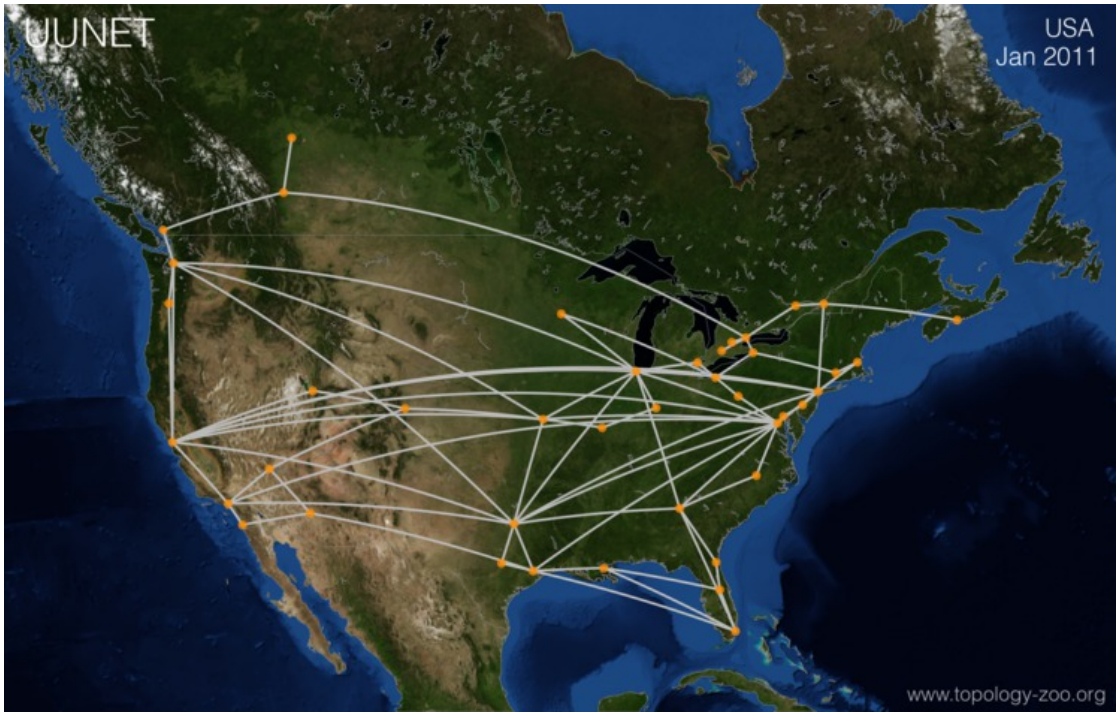


Figure 5.4: UUNET Network [24].

approximately a half of that of the Interroute Network.

Network feature	Value
number of nodes	42
number of edges	76
density	0.08
average degree	1.81
average distance	3.26
diameter	8
clustering	0.19

Table 5.10: Features of the UUNET network.

Every link of the network has been initially allocated 80 wavelengths. To keep a similar traffic load as the Interroute network, the calls request arrival process has been set with parameters $\lambda\mu = 400$, which corresponds to a network load of 21.01% and a blocking probability equal to $1.11 \cdot 10^{-3}$ in the long-run.
Given a new defined scenario, the same analysis of the 1^{st} experiment with identical methods has been repeated.

5.3.2 Numerical Results

Iterations Analysis

The analysis on the number of iterations has been again performed by varying n_{train} over the values: 10.000, 20.000, 30.000, 40.000. The results in terms of test accuracies are resumed in Table 5.11, and can be graphically observed in Fig. 5.5(A).

Accuracy	$n_{train} = 10.000$	$n_{train} = 20.000$	$n_{train} = 30.000$	$n_{train} = 40.000$
$\text{RWA}_{accuracy}$	5.30%	5.57%	5.51%	5.48%
$\text{R}_{accuracy}$	21.81%	31.99%	37.65%	39.71%
$\text{final}_{accuracy}$	63.81%	74.44%	74.86%	80.36%

Table 5.11: Iterations analysis - results for the UUNET network with 80 wavelengths. The confidence interval for the accuracies is negligible.

Except for the case of 10.000 iterations, which are not enough for this network either, all the other cases achieve similarly good performances. However, it is evident that the general curves level is lower with respect to the Interroute network, probably due to the fact that 18 rounds of messaging steps are excessive for a smaller network, as it is highlighted by the next analysis.

Messaging Steps Analysis

The analysis on the number of messaging steps has been again performed by varying n_{steps} over the values: 14, 16, 18, 20, 22. The results in terms of test accuracies are resumed in Table 5.12, and can be graphically observed in Fig. 5.5(B).

Accuracy	$n_{steps} = 14$	$n_{steps} = 16$	$n_{steps} = 18$	$n_{steps} = 20$	$n_{steps} = 22$
$\text{RWA}_{accuracy}$	5.76%	5.65%	5.57%	5.67%	5.49%
$\text{R}_{accuracy}$	51.02%	39.02%	31.99%	30.25%	35.60%
$\text{final}_{accuracy}$	90.15%	77.17%	74.44%	70.06%	72.29%

Table 5.12: Messaging steps analysis - results for the UUNET network with 80 wavelengths. The confidence interval for the accuracies is negligible.

As already anticipated in the iterations analysis, the only case able to achieve excellent performances as is $n_{steps} = 14$, meaning that the smaller the network the fewer the messaging steps needed by the MPNN in the learning process, as expected from the GNN framework theory of Chapter 4.

Learning Rate Analysis

The analysis on the learning rate has been again performed by varying l_{rate} over the values: 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}. The results in terms of test accuracies are resumed in Table 5.13, and can be graphically observed in Fig. 5.5(C).

Accuracy	$l_{rate} = 10^{-1}$	$l_{rate} = 10^{-2}$	$l_{rate} = 10^{-3}$	$l_{rate} = 10^{-4}$
$\text{RWA}_{accuracy}$	3.00%	5.57%	5.30%	4.46%
$\text{R}_{accuracy}$	13.26%	31.99%	37.40%	11.43%
$\text{final}_{accuracy}$	33.06%	74.44%	81.55%	38.65%

Table 5.13: Learning rate analysis - results for the UUNET network with 80 wavelengths. The confidence interval for the accuracies is negligible.

The results show again that the cases $l_{rate} = 10^{-3}, 10^{-2}$ represent the best trade-off to achieve good performances. However,again all the curves appear lower due to the reason explained in the previous analysis.

Batch Size Analysis

The analysis on the batch size has been again performed by varying $batch_{train}$ over the values: 25, 50, 75, 100. The results in terms of test accuracies are resumed in Table 5.14, and can be graphically observed in Fig. 5.5(D).

Accuracy	$batch_{train} = 25$	$batch_{train} = 50$	$batch_{train} = 75$	$batch_{train} = 100$
$\text{RWA}_{accuracy}$	5.51%	5.59%	5.57%	5.72%
$\text{R}_{accuracy}$	26.77%	43.85%	31.99%	39.32%
$\text{final}_{accuracy}$	70.77%	82.34%	74.44%	75.43%

Table 5.14: Batch size analysis - results for the UUNET network with 80 wavelengths. The confidence interval for the accuracies is negligible.

Whereas the best trade-off was represented by $batch_{train} = 75$ for the Interroute network, here the highest performances are achieved by $batch_{train} = 50$, meaning that for a smaller network fewer training data are needed in every iteration.

5.4 Performance Analysis: 4^{th} Experiment

5.4.1 Network Scenario and Analysis Set-Up

Given that, even by reducing the network size, the MPNN has revealed incapable of solving the RWA problem without relying on the recovery algorithm, this experiment

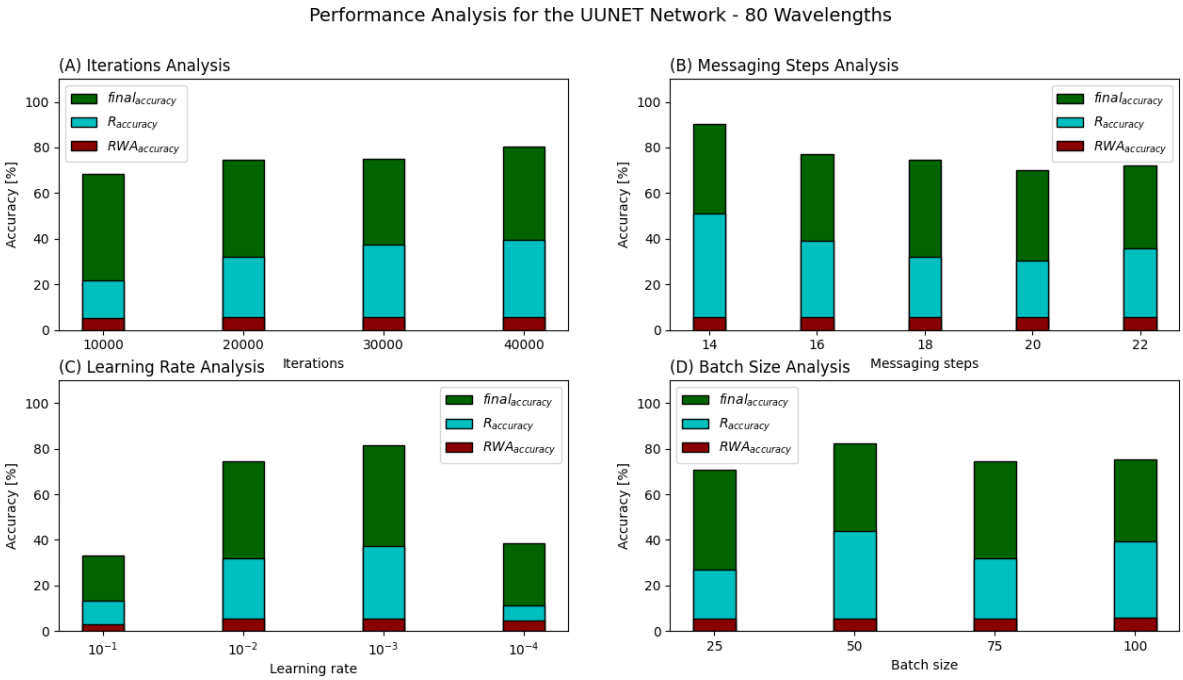


Figure 5.5: Graphical performance results for the UUNET network with 80 wavelengths.

aims at testing the MPNN within the UUNET network with only one available wavelength on every link.
The traffic load has been set up with parameters $\lambda\mu = \frac{1}{195}$, which corresponds to a network load of 0.02% and a blocking probability equal to $1.16 \cdot 10^{-3}$ in the long-run.
The same identical analysis of the 3^{rd} experiment with the same identical methods have been repeated.

5.4.2 Numerical Results

For this last experiments, the results are only reported under graphical form and can be observed in Fig. 5.6.
As already happened for the Interroute network moving from the 1^{st} to the 2^{nd} experiment, the reduction of the wavelength space from 80 to 1 has drastically increased the RWA accuracy for any hyperparameters set-up. As usual, there are some hyperparameters values which sharply represent the best trade-off in terms of accuracies, as for instance $n_{steps} = 16$, $l_{rate} = 10^{-3}$ or $batch_{size} = 75$.
Moreover, by comparing such curves with the ones of the 2^{nd} experiment, it looks

clear that all the curves are significantly higher, with some peaks even above 90% for the final accuracy. This might lead to the conclusion that adopting a small-size network with a poor wavelength space makes extremely easier for the MPNN the task of learning how to solve the RWA problem.

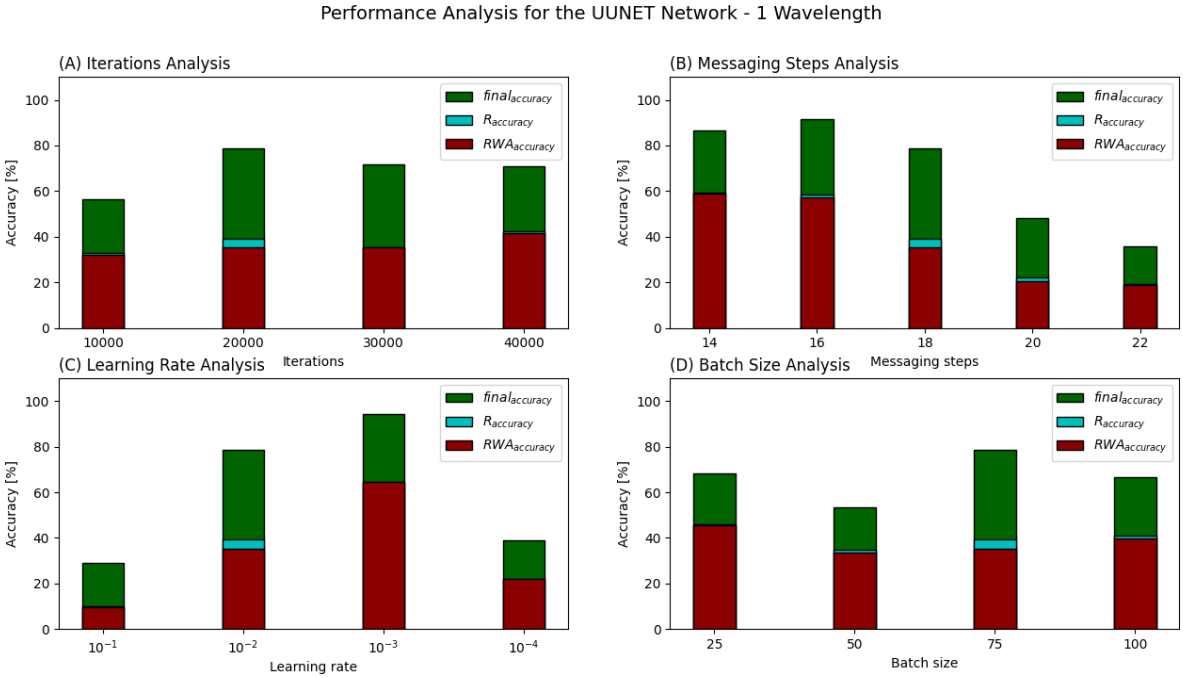


Figure 5.6: Graphical performance results for the UUNET network with 1 wavelength.

5.5 Time Complexity Analysis

In this last section of Chapter 5, a detailed overview on the framework's time complexity is presented. In particular, since the the test time is independent from the hyperparameters set-up, only the MPNN's training time is considered.
To offer a more precise analysis, the total training time for the MPNN, $time_{total}$, is divided into three main components, which are:

- $time_{data}$: the *data generation time*, which is the time needed to generate the input graphs, computing the corresponding target graphs by means of the training algorithm, and transform these data into the format suited for the framework.

- $time_{heuristic}$: the *heuristic time*, which is the time needed by the training algorithm to solve the RWA problem on every input graph.

- $time_{MPNN}$: the *MPNN training time*, which is the time needed by the MPNN to perform its total training process, as taking input data, returning an output, compute the loss, adjust the weights, etc.

Therefore, it holds: $time_{total} = time_{data} + time_{heuristic} + time_{MPNN}$.
In the following, two different time analyses are proposed. The first one aims at comparing, for a fixed network scenario, the training time depending on the hyperparameters set-up to understand how these affect the speed of the training phase; instead, the second one aims at comparing, for a fixed hyperparameters set-up, the training time for all the different network scenarios of the previous experiments to understand how the network dimension and the wavelengths space affect the speed of the learning process.

5.5.1 Hyperparameters Time Analysis

Here, the analysis of the 1^{st} experiment has been considered.

Iterations Analysis

For this analysis, the results in terms of execution times are resumed in Table 5.15, and can be graphically observed in Fig. 5.7(A).

Time	$n_{train} = 10.000$	$n_{train} = 20.000$	$n_{train} = 30.000$	$n_{train} = 40.000$
$time_{total}$	8h 51m	17h 53m	28h 32m	35h 37m
$time_{data}$	5h 22m	10h 50m	17h 24m	21h 46m
$time_{heuristic}$	1h 36m	3h 04m	5h 15m	5h 50m
$time_{MPNN}$	1h 53m	3h 50m	5h 50m	8h 01m

Table 5.15: Iterations time analysis - results for the Interroute network with 80 wavelengths. The confidence interval for the execution times is negligible.

As expected, the training time is perfectly linear versus the number of iterations. Moreover, one can also notice that the predominant component of the training time is dedicated to data generation (61%), whereas the other two components are approximately equivalent and around 20%. This percentage sub-division holds true in general for all the following time analyses.

Messaging Steps Analysis

For this analysis, the results in terms of execution times are resumed in Table 5.16, and can be graphically observed in Fig. 5.7(B).

Time	$n_{steps} = 14$	$n_{steps} = 16$	$n_{steps} = 18$	$n_{steps} = 20$	$n_{steps} = 22$
$time_{total}$	17h 01m	17h 16m	17h 53m	18h 03m	18h 24m
$time_{data}$	10h 50m	10h 50m	10h 50m	10h 50m	10h 50m
$time_{heuristic}$	3h 04m	3h 04m	3h 04m	3h 04m	3h 04m
$time_{MPNN}$	3h 06m	3h 21m	3h 50m	4h 08m	4h 29m

Table 5.16: Messaging steps time analysis - results for the Interroute network with 80 wavelengths. The confidence interval for the execution times is negligible.

As expected, the only component which is affected by the number of messaging steps is $time_{MPNN}$, which slightly increases as n_{steps} increases.

Learning Rate Analysis

For this analysis, the results in terms of execution times are resumed in Table 5.17, and can be graphically observed in Fig. 5.7(C).

Time	$l_{rate} = 10^{-1}$	$l_{rate} = 10^{-2}$	$l_{rate} = 10^{-3}$	$l_{rate} = 10^{-4}$
$time_{total}$	17h 37m	17h 53m	17h 44m	17h 47m
$time_{data}$	10h 50m	10h 50m	10h 50m	10h 50m
$time_{heuristic}$	3h 04m	3h 04m	3h 04m	3h 04m
$time_{MPNN}$	3h 34m	3h 50m	3h 41m	3h 44m

Table 5.17: Learning rate time analysis - results for the Interroute network with 80 wavelengths. The confidence interval for the execution times is negligible.

Again, the only component affected by the l_{rate} is $time_{MPNN}$, although the times are similar for any value of such hyperparameter.

Batch Size Analysis

For this analysis, the results in terms of execution times are resumed in Table 5.18, and can be graphically observed in Fig. 5.7(D).

As for the iterations case, the training time looks perfectly linear versus the batch size, as expected.

Time	$batch_{train} = 25$	$batch_{train} = 50$	$batch_{train} = 75$	$batch_{train} = 100$
$time_{total}$	6h 28m	13h 00m	17h 53m	24h 37m
$time_{data}$	3h 51m	7h 43m	10h 50m	14h 53m
$time_{heuristic}$	1h 07m	2h 04m	3h 04m	4h 00m
$time_{MPNN}$	1h 30m	3h 13m	3h 50m	5h 44m

Table 5.18: Batch size time analysis - results for the Interroute network with 80 wavelengths. The confidence interval for the execution times is negligible.

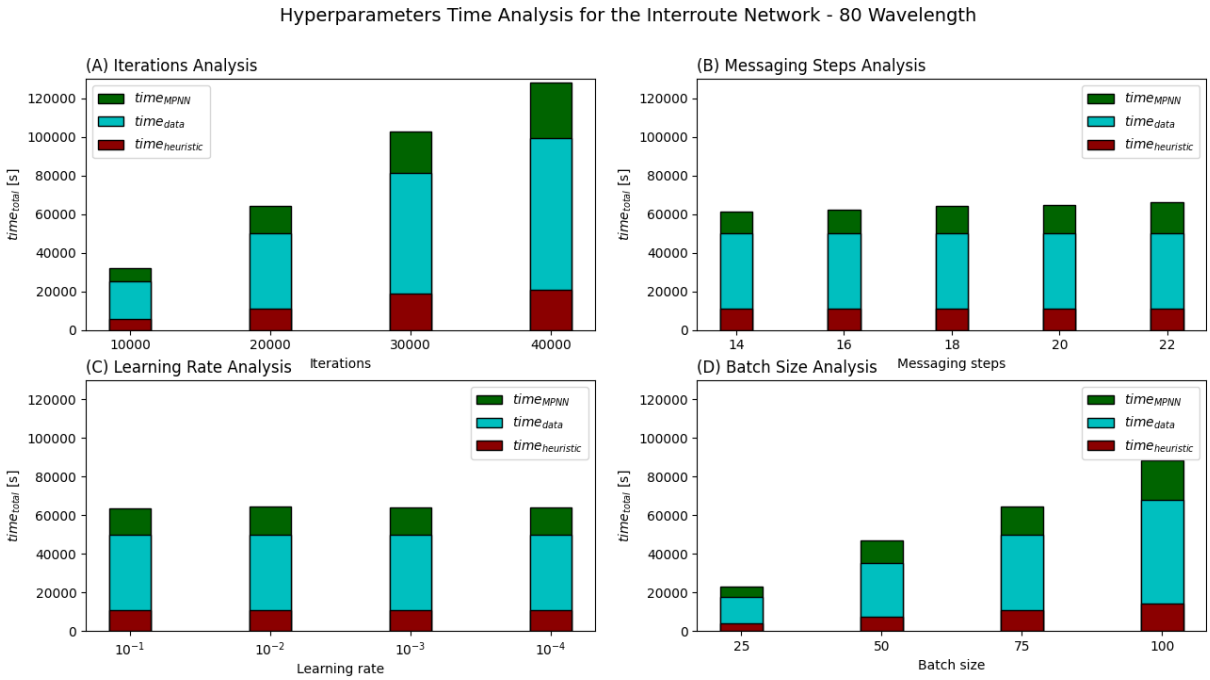


Figure 5.7: Graphical hyperparameters time analysis results for the Interroute network with 80 wavelengths.

5.5.2 Network Scenarios Time Analysis

Here, the analysis with the all the hyperparameters set to the default values have been considered for each of the four network scenarios. The results are resumed in Table 5.19, and can be graphically observed in Fig. 5.8.

The results highlight that the MPNN's training time, for a fixed set of hyperparameters, is strongly related to both the network size and the number of considered wavelengths. In fact, by moving from a medium-large-size network as the Interroute to a small-size network as the UUNET, the training time decreases by 38%, due to

Time	UUNET#1	UUNET#80	Interroute#1	Interroute#80
$time_{total}$	6h 05m	11h 13m	9h 31m	17h 53m
$time_{data}$	3h 42m	6h 48m	6h 18m	10h 50m
$time_{heuristic}$	0h 28m	1h 57m	0h 46m	3h 04m
$time_{MPNN}$	1h 55m	2h 28m	2h 27m	3h 50m

Table 5.19: Network scenarios time analysis. The confidence interval for the execution times is negligible.

the fact that the possible routes are and shorter and in lower amount. Moreover, given the same network, reducing the wavelength space from 80 to 1 leads to a training time which is almost half of the original one, due to the faster wavelength search.
In conclusion, one can firmly state that the MPNN's training time for the RWA problem varies linearly as the network dimension and the wavelength resources vary.

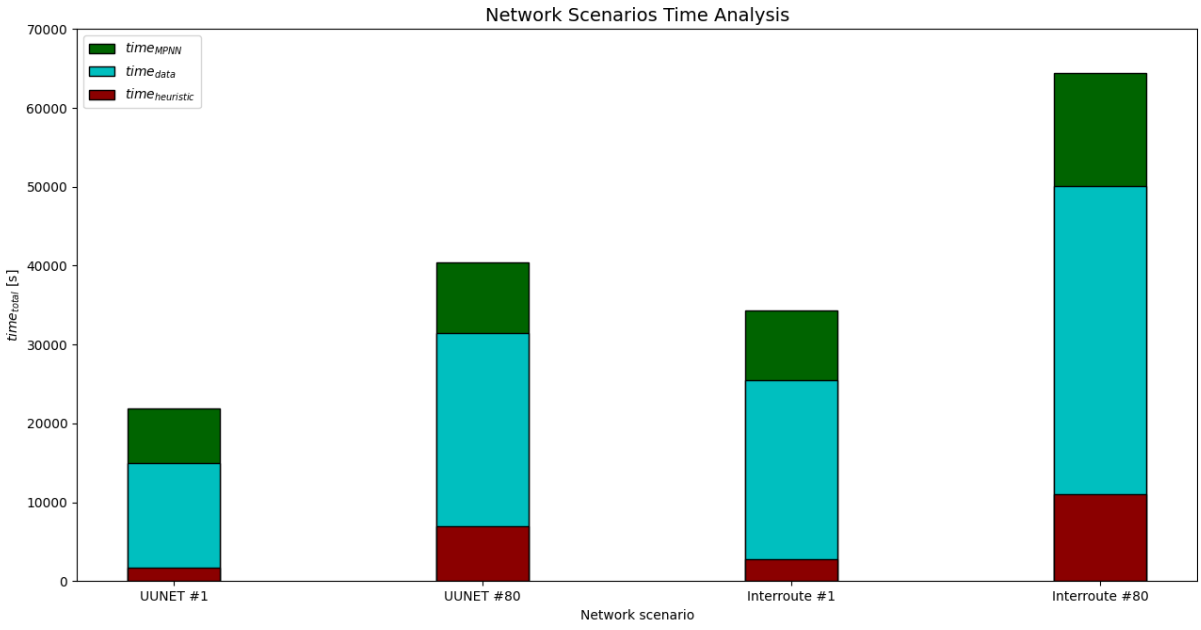


Figure 5.8: Graphical network scenarios time analysis results. All the hyperparameters are set to the default values.

Chapter 6

Conclusion

6.1 Final Considerations

The thesis project, entitled *Machine Learning-Based Routing and Wavelength Assignment in Optical Networks*, has been developed with the purpose of applying the methods of an innovative AI area, which is ML GNNs, to a traditional networking problem, which is RWA.
Having introduced the RWA problem and how this can be optimally solved by an ILP formulation, in case of SLE, or sub-optimally solved by heuristic approaches, in case of DLE, it has been discussed that such problem belongs to the NP-complete class.
Therefore, the starting point of the project has been investigating whether the NP-complete nature of the RWA problem could be bypassed by leveraging a non-traditional paradigm. A paradigm which does not require any human-developed rules, but finds the rules itself by simply analyzing the available labeled data. Such an innovative idea has been implemented by means of GNNs, a particular type of NNs able to exploit graph-based problems' structures. In fact the whole project has been focused on a powerful framework able to build a particular type of GNN, which is the MPNN. This new architecture has been trained to learn how to solve the RWA problem within a specific network scenario, and finally tested to obtain quantitative measures of the achievable performances.
The results have shown that it is possible to find the proper MPNN's parameters set-up and hence achieving excellent performances, with accuracies even above 90%. Therefore, when the results are satisfactory, the MPNN could hypothetically be implemented within a realistic network scenario, and offer a reliable alternative to traditional algorithms for the RWA problem. In conclusion, such results have demonstrated that the networking area is definitely suited to face some of its problems by means of ML methods able to offer performances, flexibility and

scalability.

6.2 Future Works

The thesis has offered interesting results, but has unarguably been limited in the amount of computational resources. Therefore, it can be relevant to let the reader reason about some future works that could explore the whole potential of this project.
First, extending the analysis to an even larger set of hyperparmeters could lead to even higher performances than the achieved ones. For instance, it would be interesting to find a precise relationship between the network size and the number of messaging steps required. In addition, future works could consist in performing these analysis on very large networks with a large wavelengths space in order to see how scalable the framework is. Moreover, it would be worth to try adopting different training algorithms, such as an adaptive routing followed by a different WA heuristic than the First-Fit. Finally, the GNNs represent in general a powerful tool to work on graph-based structures, and hence different architectures from the MPNN could be tested for the RWA problem.

Bibliography

[1] Thomas Erlebach and Klaus Jansen. «The complexity of Path Coloring and Call Scheduling». In: *Theoretical Computer Science 255 33–50* (2001) (cit. on p. 2).

[2] Hui Zang, Jason P. Jue, and Biswanath Mukherjee. «A Review of Routing and Wavelength Assignment Approaches for Wavelength-Routed Optical WDM Networks». In: *OPTICAL NETWORKS MAGAZINE* (January 2000) (cit. on pp. 3–5, 8, 10, 11).

[3] Ori Gerstel and Shay Kutten. «Dynamic Wavelength Allocation in All-optical Ring Networks». In: *Proc., IEEE ICC '97* (June 1997) (cit. on pp. 3, 5).

[4] S. Even, A. Itai, and A. Shamir. «On the Complexity of Timetable and Multicommodity Flow Problems». In: *SIAM Journal of Computing* (1976) (cit. on p. 6).

[5] I. Martìn, S. Troia, J.A. Hernàndez, A. Rodrìguez, F. Musumeci, G. Maier, R. Alvizu, and O. Gonzàlez de Dios. «Machine Learning-Based Routing and Wavelength Assignment in Software-Defined Optical Networks». In: *IEEE Transactions on Network and Service Management* (2019) (cit. on pp. 6, 12, 13).

[6] Wikipedia contributors. *Routing and Wavelength Assignment — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/wiki/Routing_and_wavelength_assignment`. [Online; accessed 9-February-2021]. 2021 (cit. on p. 7).

[7] I. Szczesniak, A. Jajszczyk, and B. Wozna-Szczesniak. «Generic Dijkstra for Optical Networks». In: *Journal of Optical Communications and Networking* (2019) (cit. on p. 10).

[8] R. Ramaswami and K. Sivarajan. «Routing and Wavelength Assignment in All-Optical Networks». In: *IEEE/ACM TRANSACTIONS ON NETWORKING* (1995) (cit. on p. 11).

[9] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. «A Machine Learning Approach to Routing». In: *arXiv:1708.030747* (2017) (cit. on pp. 13, 14).

[10] Zhizhen Zhong, Nan Hua, Zhigang Yuan, Yanhe Li, and Xiaoping Zheng. «Routing without Routing Algorithms: An AI-Based Routing Paradigm for Multi-Domain Optical Networks». In: *OFC 2019* (2019) (cit. on p. 13).

[11] Xiaoliang Chen, Jiannan Guo, Zuqing Zhu, Roberto Proietti, Alberto Castro, and S. J. B. Yoo. «Deep-RMSA: A Deep-Reinforcement-Learning Routing, Modulation and Spectrum Assignment Agent for Elastic Optical Networks». In: *OFC 2018* (2018) (cit. on p. 13).

[12] C. Natalino, M.R. Raza, P. Ohlen, P. Batista, M. Santos, L. Wosinska, and P. Monti. «Machine Learning based Routing of QoS Constrained Connectivity Services in Optical Networks». In: *OSA 2018* (2018) (cit. on p. 13).

[13] Zirui Zhuang, Jingyu Wang, Qi Qi, Haifeng Sun, and Jianxin Liao. «Graph-Aware Deep Learning Based Intelligent Routing Strategy». In: *IEEE 43rd Conference on Local Computer Networks* (2018) (cit. on p. 13).

[14] Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. «Neural Execution of Graph Algorithms». In: *33rd Conference on Neural Information Processing Systems* (2019) (cit. on p. 13).

[15] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. «Graph Neural Networks: A Review of Methods and Applications». In: *arXiv:1812.08434* (2019) (cit. on pp. 13, 27).

[16] Wikipedia contributors. *Machine Learning — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/wiki/Machine_learning#Machine_learning_approaches`. [Online; accessed 15-February-2021]. 2021 (cit. on pp. 14, 18).

[17] Ravil Muhamedyev. «Machine LEarning Methods: An Overview». In: *COMPUTER MODELLING NEW TECHNOLOGIES* (2015) (cit. on p. 16).

[18] Vladislav Skorpil and Jiri Stastny. «Neural Networks and Back Propagation Algorithm». In: *ELECTRONICS' 2006* (2006) (cit. on pp. 17, 18).

[19] Peter W. Battaglia et al. «Relational inductive biases, deep learning, and graph networks». In: *arXiv:1806.01261* (2018) (cit. on pp. 19–23, 26–29).

[20] Peter Roelant. *Softmax classification with cross-entropy*. `https://peterroelants.github.io/posts/cross-entropy-softmax/`. [Online; accessed 26-February-2021]. 2019 (cit. on p. 40).

[21] Diederik Kingma and Jimmy Ba. «Adam: A Method for Stochastic Optimization». In: *arXiv:1412.6980* (2015) (cit. on p. 40).

[22] Youkabed Sadri. «Fast and scalable routing in large telecommunication networks». In: *Politecnico di Torino, Master Degree Thesis* (2020) (cit. on p. 60).

[23] *Computational resources provided by HPC@POLITO, which is a project of Academic Computing within the Department of Control and Computer Engineering at the Politecnico di Torino*. `http://www.hpc.polito.it`. [Online]. 2021 (cit. on p. 63).

[24] *The Internet Topology Zoo*. `http://www.topology-zoo.org/index.html`. [Online]. 2021 (cit. on pp. 64, 71, 72).