UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Anomaly Detection in Microservices Using Execution Traces and Graph Neural Networks

Versão final após defesa

Sara Maria da Silva Martins

Dissertação para a obtenção do grão de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Professor Doutor Mário Marques Freire

Janeiro 2025

Declaração de Integridade

Eu, Sara Maria da Silva Martins, que abaixo assino, estudante com o número de inscrição M13382 de Engenharia Informática da Faculdade de Engenharia, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o **Código de Integridades da Universidade da Beira Interior**.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 11/01/2025

Dissertation prepared at Secure and Intelligent Networked Software Systems Laboratory (sins lab), home of the Network Applications and Services - Cv Group of the Instituto de Telecomunicações and at the Department of Computer Science of the University of Beira Interior and submitted to the University of Beira Interior for discussion in public session to obtain the M. Sc. Degree in Computer Science and Engineering.

Acknowledgements

Fátima and Abílio, for being the most extraordinary parents I could ever ask for. Your support in this five year long journey was paramount for me to be here today.

Catarina, for being the greatest, most incredible big sister anyone could ask for. I am truly in your debt for everything I have learned from you.

Mirki and Lumi, for lighting up even the darkest days even when cuddles become scratches.

Jorge, for all the support in the roller-coaster of emotions that were these years together. To many, many more.

Ana, for not even distance nor mismatched schedules can ever separate us in the end. To another twenty years of friendship.

Margarida, for being the most pleasant surprise I could have found here. University felt lighter with you.

Euan and Tristan, for their patience when being my personal consultants on writing the real English.

To the boys, especially Alexandre, Duarte, and Manuel, for the amazing five years of university that now end. This journey would be way worse without you.

Professors of the Informatics and Mathematics departments, especially mentioning Pedro Inácio, Isabel Cunha, and Gastão Bettencourt, who accompanied me in this academic experience.

Professor Mário Freire, for being the always supportive, patient, and nice adviser he has been these last years.

Informatics Department and Instituto de Telecomunicações, for the computer lended for the execution of the computational part of the project.

Daniele Grattarola, for the help with the Spektral library.

Resumo

Os métodos tradicionais de deteção de anomalias baseiam-se muitas vezes na análise manual de *logs* e são insuficientes para as interações complexas e de múltiplas camadas dos microsserviços. A existência de múltiplas chamadas de e para os mesmos serviços, seguindo diferentes caminhos de execução, é difícil de acomodar nos métodos tradicionais, usualmente lineares. A deteção de anomalias desempenha um papel fundamental na manutenção da saúde dos sistemas e no cumprimento de *Service-Level Agreements* (SLAs), identificando comportamentos invulgares que podem indicar falhas no sistema ou vulnerabilidades de segurança.

Esta insuficiência dos métodos tradicionais pode implicar a não deteção de falhas em sistemas de microsserviços. Não conseguir monitorizar corretamente um sistema pode traduzir-se em falhas graves e potencialmente danosas, pelo que é necessário encontrar forma de detetar os novos tipos de anomalias.

Este trabalho de investigação propõe um modelo de Rede Neuronal de Grafos (RNG), com diferentes camadas convolucionais e funções de ativação para otimizar o desempenho, utilizando o *dataset* AIOps Challenge 2020 para avaliação. Os resultados mostram que os modelos mais simples com menos camadas escondidas alcançam o melhor desempenho, ultrapassando frequentemente a *accuracy* de 99.9%. A investigação conclui que os métodos de deteção de anomalias devem ser adaptados a sistemas específicos e que as RNGs oferecem uma abordagem promissora para lidar com as complexidades das arquiteturas de microsserviços.

Palavras-chave

Aprendizagem Automática, Desafio AIOps 2020, Deteção de Anomalias, Microsserviços, Redes Neuronais de Grafos, Traços de Execução.

Resumo Alargado

O foco principal desta dissertação é o desenvolvimento de um algoritmo para deteção de anomalias em arquiteturas de microsserviços, utilizando técnicas de Aprendizagem Automática (AA), nomeadamente, Rede Neuronal de Grafos (RNG). A investigação aborda a transição de sistemas monolíticos tradicionais para arquiteturas distribuídas baseadas em microsserviços, que oferecem maior flexibilidade, mas também aumentam a complexidade em termos de manutenção, segurança e monitorização de desempenho.

Os sistemas monolíticos, onde todos os componentes estão consolidados num único bloco de código, evoluíram para arquiteturas orientadas a serviços e microsserviços. Estes sistemas distribuídos dividem a aplicação em unidades menores e independentes que comunicam entre si através de *Application Programming Interfaces* (APIs). Embora esta arquitetura ofereça vantagens como escalabilidade e disponibilidade, também apresenta novos desafios como a gestão de estados, distribuição geográfica e consistência de dados. Uma das principais dificuldades que as organizações enfrentam após a migração para microsserviços é garantir a segurança e manutenção contínua destes sistemas.

Neste contexto, a deteção de anomalias desempenha um papel crucial na manutenção da saúde do sistema e no cumprimento de *Service-Level Agreements* (SLAs). As anomalias referem-se a comportamentos inesperados do sistema que podem indicar falhas ou vulnerabilidades de segurança. Estas anomalias podem manifestar-se de diversas formas e em diferentes dimensões, pelo que detetá-las celeremente é essencial para evitar tempos de inatividade, perda de dados ou danos à reputação.

Os métodos atuais de deteção de anomalias em microsserviços costumam depender da análise de *logs*, de forma manual ou com recurso a ferramentas de visualização, exigindo um conhecimento profundo do sistema e das potenciais anomalias. No entanto, as abordagens tradicionais de deteção de anomalias em arquiteturas monolíticas ou orientadas a serviços não contemplam na totalidade a complexidade dos microsserviços, que envolvem múltiplas cadeias de chamadas e interações entre serviços independentes. Esta complexidade requer mecanismos de deteção de anomalias mais avançados que considerem a natureza multidimensional das falhas em microsserviços.

Vários algoritmos foram propostos para a resolução deste problema, com diferentes níveis de precisão. Contudo, ainda existem lacunas importantes, incluindo a ausência de um método universal que lide com todos os tipos de anomalias e a necessidade de grandes volumes de dados para obter resultados precisos. Além disso, os métodos existentes costumam falhar em capturar todas as possíveis cadeias de chamadas dentro de um sistema.

Para enfrentar estes desafios, esta dissertação propõe o uso de RNG para detetar anomalias em sistemas de microsserviços. As RNG são particularmente adequadas para esta tarefa de-

vido à sua capacidade de modelar relacionamentos e interações complexas entre os serviços. O objetivo é desenvolver um modelo baseado em RNG que seja não apenas eficaz na deteção de anomalias, mas também adaptável e escalável em diferentes ambientes de microsserviços.

São necessários métodos mais eficientes para a deteção de anomalias em microsserviços, onde desvios no tempo de execução são usados para detetar problemas de desempenho. Este projeto assume que os tempos de execução dos serviços permanecem relativamente consistentes, o que permite o uso dessa informação para a deteção de anomalias. No entanto, uma das questões que ainda se coloca é a falta de um método de deteção universal que possa ser aplicado a diferentes sistemas e tipos de anomalias.

O principal objetivo desta investigação é conceber um modelo de RNG, com recurso à biblioteca de Python `Spektral`, experimentando diferentes tipos e números de camadas convolucionais escondidas para obter o melhor desempenho. O objetivo passa por ultrapassar as soluções existentes no estado da arte e tornar o modelo o mais adaptável possível a diferentes sistemas de microsserviços.

As questões de investigação que orientam este trabalho são:

- Como melhorar a *accuracy* da deteção de anomalias no sistema de microsserviços representado pelo *dataset* do AIOps Challenge 2020 utilizando RNG?
- Qual o número e tipo de camadas convolucionais que conduzem aos melhores resultados de classificação?

O projeto segue a *Design Science Research Methodology* (DSRM), uma abordagem estruturada em seis etapas para trabalhos de investigação:

1. Identificação do Problema e Motivação;
2. Definição de Objetivos;
3. Conceptualização e Desenvolvimento;
4. Demonstração;
5. Avaliação;
6. Comunicação.

A principal contribuição desta investigação é o desenvolvimento de um algoritmo baseado em RNG para deteção de anomalias em sistemas de microsserviços utilizando traços de execução. O código do projeto será disponibilizado no GitHub[1] e a dissertação será submetida

[1] `https://github.com/pipademus/`

ao arquivo digital da University of Beira Interior (UBI) para referências futuras.

A arquitetura de microsserviços é um paradigma moderno de *design* de *software*, onde aplicações são construídas como serviços independentes, cada qual responsável por uma função específica. Esta arquitetura permite escalabilidade, isolamento de falhas e flexibilidade, mas também traz desafios, como o aumento da complexidade do sistema e dificuldades na gestão de dependências. Estas complexidades tornam a deteção de anomalias – a identificação de padrões anormais no comportamento do sistema – crucial para manter o desempenho e a confiabilidade de aplicações baseadas em microsserviços.

Esta dissertação estuda várias abordagens do estado da arte para deteção de anomalias, divididas em três tipos:

- Baseada em Regras – inclui *frameworks* como *Statistical Learning-Based Metric Anomaly Detection* (SLMAD) e TraceRCA, que dependem de métodos estatísticos, regras codificadas manualmente e técnicas baseadas em gráficos para detetar anomalias;

- AA – abrange métodos como *Adaptive Label Screening and Relearning* (ALSR) e iR-RCF que utilizam modelos, por exemplo, *Random Forest* (RF), XGBoost e aprendizagem ativa, para se adaptar às mudanças no comportamento do sistema sem a necessidade de definição manual de regras;

- *Deep Learning* (DL) – engloba modelos como TraceGra, TELESTO e DeepTraLog, que utilizam técnicas avançadas de redes neuronais para detetar anomalias complexas.

O *dataset* utilizado neste projeto vem do concurso AIOps Challenge 2020, composto por mais de dezanove milhões de *logs* que representam *spans* dentro de um sistema de microsserviços. Estes *spans* foram transformados em grafos, onde os nós representam *spans* individuais e as arestas simbolizam as conexões entre eles, com base nas características selecionadas.

A configuração do modelo passa pela utilização de uma, cinco, ou dez camadas escondidas, de um de oito tipos de camadas convolucionais: `AGNNConv`, `EdgeConv`, `GATConv`, `GeneralConv`, `GINConv`, `GraphSageConv`, `GTVConv` e `TAGConv`. Em cada camada é utilizada uma de quatro funções de ativação possíveis: *Rectified Linear Unit* (ReLU), sigmoide, *soft-plus* e tangente hiperbólica (TanH).

O desempenho é avaliado por métricas estatísticas como *accuracy*, *f-score*, precisão, e *recall*, bem como por métricas de consumo, como tempo, *Central Processing Unit* (CPU) e *Random Access Memory* (RAM). Uma abordagem sistemática garante que os procedimentos de treino e teste são consistentes e padronizados durante as diferentes execuções.

Os resultados revelam que os modelos atingiram, em geral, uma precisão elevada, quase sempre superior a 99%. A função de ativação *soft-plus* apresentou consistentemente o melhor desempenho. O aumento do número de camadas escondidas nem sempre se traduziu numa

melhoria das métricas de avaliação, levando apenas a um maior consumo de recursos computacionais. Os modelos com uma única camada escondida, de tipo `EdgeConv` ou `TAGConv`, combinadas com a função de ativação *soft-plus*, apresentaram o melhor desempenho global.

As conclusões finais refletem a natureza única de cada sistema e algoritmo. Os métodos de deteção de anomalias devem ser adaptados ao sistema específico e os algoritmos de AA devem ser testados exaustivamente para determinar a sua adequação. A investigação salienta a importância de compreender as arquiteturas dos sistemas, de modo especial para conseguir implementar o melhor método de deteção de anomalias para cada caso único.

Abstract

Traditional anomaly detection methods are often based on manual analysis of logs and are insufficient for the complex, multi-layered interactions of microservices. The existence of multiple calls to and from the same services, following different execution paths, is difficult to accommodate in traditional, usually linear, methods. Anomaly detection plays a key role in maintaining systems health and complying with Service-Level Agreements (SLAs) by identifying unusual behavior that may indicate system failures or security vulnerabilities.

This shortcoming of traditional methods can mean that faults are not detected in microservice systems. Failure to properly monitor a system can lead to serious and potentially damaging failures, thus, to find ways of detecting new types of anomalies is imperative.

This research work proposes a Graph Neural Network (GNN) model, with different convolutional layers and activation functions to optimize performance, using the AIOps Challenge 2020 dataset for evaluation. The results show that simpler models with fewer hidden layers achieve the best performance, often exceeding the accuracy of 99.9%. The research concludes that anomaly detection methods should be tailored to specific systems and that GNNs offers a promising approach to dealing with the complexities of microservice architectures.

Keywords

AIOps Challenge 2020, Anomaly Detection, Execution Traces, Graph Neural Networks, Machine Learning, Microservices.

Contents

List of Figures

List of Tables

Acronyms

AA Aprendizagem Automática

ACM Association for Computing Machinery

AE Auto-Encoder

AI Artificial Intelligence

ALSR Adaptive Label Screening and Relearning

API Application Programming Interface

CCS Computing Classification System

CNN Convolutional Neural Network

CPU Central Processing Unit

DAG Directed Acyclic Graph

DL Deep Learning

DNN Deep Neural Network

DOMA Domain-Oriented Microservice Architecture

DQM Data Quality Monitor

DSRM Design Science Research Methodology

GAT Graph Attention Network

GCNN Graph Convolutional Neural Network

GGNN Gated Graph Neural Network

GMTA Graph-Based Approach of Microservice Trace Analysis

GNN Graph Neural Network

GPU Graphics Processing Unit

GVAE Graph Variational Auto-Encoders

IBM International Business Machines Corporation

KNN K-Nearest Neighbours

KPI Key Performance Indicator

LSTM Long Short-Term Memory

MEPFL Microservice Error Prediction and Fault Localisation

ML Machine Learning

MLP Multi-Layer Perceptron

MMD Moving Metric Detector

PCA Principal Component Analysis

RAM Random Access Memory

RCA Root Cause Analysis

ReLU Rectified Linear Unit

RF Random Forest

RNG Rede Neuronal de Grafos

RRCF Robust Random Cut Forest

RS Request String

RWDG Request Weighted Directed Graph

SGVB Stochastic Gradient Variational Bayes

SLA Service-Level Agreement

SLMAD Statistical Learning-Based Metric Anomaly Detection

SLR Systematic Literature Review

SOA Service-Oriented Architecture

SR Spectral Residual

SVDD Support Vector Data Description

TanH Hyperbolic Tangent

TRL Technology Readiness Level

UBI University of Beira Interior

URL Uniform Resource Locator

VAE Variational Auto-Encoder

VGAE Variational Graph Auto-Encoder

Chapter 1

Introduction

1.1 Scope and Motivation

The first computer systems were implemented in the monolithic architecture, in which the entirety of a system is concentrated on a single block of code. The emergence of distributed architectures, namely Service-Oriented Architecture (SOA) and the microservices architecture, allow for a decentralisation of the code. This brings greater flexibility at the expense of greater complexity, as will be further discussed in section 2.2 [4].

Even so, in most cases the advantages outweigh the disadvantages and many organisations have dedicated themselves to migrating their infrastructure from a monolithic architecture to an SOA or microservices architecture. This migration intends to help organisations guarantee the high scalability and high availability requirements of the Service-Level Agreement (SLA) [5, 6].

Although migration itself involves its own challenges, like multi-tenancy, statefulness, and data consistency, one of the main challenges organisations face after its completion is the maintenance and security of their systems [6]. In a monolith architecture, security can be done *in situ*, considering only a single block of code and a single way of processing requests and information. In a distributed architecture, several blocks of code, varying call chains and mutable processes must be taken into account [2, 7].

Services are usually accompanied by SLAs, which define the minimum standards that must be guaranteed. If the requirements are not met, then data, assets, and reputation can be lost [8]. Constantly monitoring systems is therefore of major importance to avoid this situation, in particular to detect any potential anomalies that may occur [9].

Anomaly detection can be defined as the process of identifying a series of observations that most likely are not representative of the normal behaviour of a system [10, 11]. These can be divided into four main dimensions, which will be presented in depth in section 2.3: node, instance, configuration, and sequence [9].

One of the greatest challenges faced by the microservice architectural design is the detection of failures in a timely manner. In practice, anomaly detection depends on manual log analysis, sometimes aided with visualisation and trace analysis, requiring professionals with extensive experience and knowledge. Existing monolithic or service-oriented anomaly detection approaches do not consider nor support the multi-dimensional nature of microservice

faults [9].

Several algorithms have since then been proposed, as shown in table 1.1. These represent the works which present their results with multiple metrics. For example, the work which achieves the best accuracy value, 98.3, is not shown here as it only considers one metric. The long list of submitted works is available in table 2.3, under subsection 2.4.3.

Table 1.1: Overview of selected state-of-the-art algorithms considered for this dissertation on anomaly detection, ordered by year and name of the author. A: Accuracy, F: F1-Score, P: Precision, R: Recall.

Year	Works	Performance
2019	Zhou et al. [12]	F: 92.7%, P: 98.6%, R: 87.4%
2020	Arsalan et al. [13]	F: 64.6%, P: 85.5%, R: 56.5%
2021	Chen et al. [14]	A: 91.5%, F: 88.6%, R: 89.0%
	Li et al. [15]	F: 80.0%, P: 85.0%, R: 80.0%
	Scheinert et al. [16]	A: 85.1%, F: 85.4%, P: 86.1%, R: 85.1%
2022	Zhang et al. [17]	F: 95.4%, P: 93.0%, R: 97.8%
2023	Chen et al. [18]	F: 95.3%, P: 97.1%, R: 93.5%
	Zeng et al. [19]	F: 95.3%, P: 95.0%, R: 95.8%

Table 1.1 presents some of the results obtained for anomaly detection in terms of Accuracy, F-Score, Precision, and Recall, common performance measures which will be further studied in section 3.5. These papers also point some issues which need solving [17, 18, 20, 21]:

- The lack of an accurate universal anomaly detection method;
- The wide range of possible anomalies which need covering;
- The data must be big enough to provide accurate results, thus a system utilising less data to achieve better results is desirable;
- The representation of all possible chain calls within a system.

This project is developed with these gaps in mind, working with Graph Neural Networks (GNNs) to provide a solution for timely anomaly detection in microservices systems. This work aims to investigate an algorithm to perform anomaly detection in a microservice architecture system, utilising knowledge of Machine Learning (ML), programming, and neural networks. The primary areas of relevance according to the Association for Computing Machinery (ACM) Computing Classification System (CCS) are [22]:

- Computer Systems Organisation:
  - Dependable and Fault-Tolerant Systems and Networks;
- Computing methodologies:

  - Machine Learning Approaches;
  - Neural Networks.

1.2 Problem Statement and Objectives

Initial anomaly detection approaches use execution time deviations in order to detect performance anomalies. That makes it necessary to consider how the different call chains impact each execution and call times. This project assumes that the execution time of each service does not deviate significantly depending on the different calls, due to the unlikeliness of two or more different functionalities within each service [23, 24, 25].

One of the issues still faced by anomaly detection methods, both in the academia and in the industry, is the lack of a universal anomaly detection method [20]. Microservices are small, independent units of code with complex interactions between them, each of which performing a unique function. A process can traverse several microservices to carry out an action, so a call chain must be considered for each individual process. An anomaly in the call chain may not be detected immediately when it occurs, so constant monitoring of the system is necessary to speed up and accelerate this detection process.

This project aims at providing a solution for the anomaly detection problem faced by the microservice architecture, here represented in the AIOps Challenge 2020 dataset, while endeavouring to be better than the state-of-the-art solutions. Other solutions, reviewed in section 2.4, leverage various algorithms, but none of the studied papers applied GNNs for anomaly detection. Thus, this work focuses on applying the `Spektral` Python library on GNNs with different number and type of convolutional layers, explained further in section 3.4, to find the combination which provides the best results [26].

The main objective is to research a GNN built with the `Spektral` Python library that is not only capable of surpassing the existing state-of-the-art solutions, but can also be easily applied to different microservice systems. The intermediate objectives, related to the approach chosen for this project, are the following:

- Contextualise the necessary theoretical bases, acquiring concepts about microservice architecture, anomaly detection, and neural networks;
- Identify features of the chosen dataset suitable for anomaly detection;
- Design, implement, and validate a GNN capable of performing anomaly detection in microservices;
- Modify the GNN to be as universal and adaptable as possible;

- Within time constraints, further develop the algorithm to execute Root Cause Analysis (RCA).

Following the objectives previously outlined, this dissertation aims to answer the following research questions:

> Research Question 1: How to better the anomaly detection accuracy in the microservices system represented by the AIOps Challenge 2020 dataset using GNN?
>
> Research Question 2: Which number of hidden layers, and which type of convolutional layer, lead to the best classification results?

1.3 Approach

The Design Science Research Methodology (DSRM) is a six-step process which defines a standardised approach to research work, as shown in figure 1.1 [1]. Its steps can be defined as follows[1]:

1. Problem Identification and Motivation – highlight the specific research problem, and justify the need for a solution in addition to those already existing;

2. Define the Objectives for a Solution – deduce the objectives to be achieved from the defined problem, considering the knowledge of what is possible and feasible;

3. Design and Development – create the artifact containing the contribution or solution to the problem, with a prior definition of the architecture and functionalities required. The AIOps Challenge 2020 dataset, discussed further in section 3.2, was chosen and studied to train, validate, and test the GNN model, which is the artifact of this project;

4. Demonstration – demonstrate how the artifact solves the problem or contributes to its resolution, by comparing it to a baseline algorithm;

5. Evaluation – measure and analyse the effectiveness and efficiency of the artifact, comparing the observed results with the defined objectives;

6. Communication – communicate the problem, its relevance, the artifact, and the results achieved to the target audience (scholars, researchers, or professionals).

[1]The steps are numbered in the order presented by the researchers; however, it should be noted that during the course of the work the execution of these steps may not be sequential. This is recognised in the article, as the research process can be irregular.

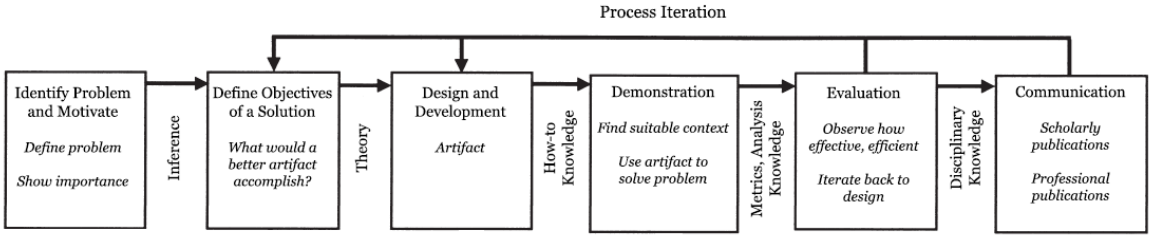

Figure 1.1: The six steps of the DSRM process model (adapted from [1]).

1.4 Main Contributions

The objectives outlined in section 1.2 define the creation of a GNN model for detecting anomalies in microservice systems through the use of execution traces. The method applied is the main contribution accomplished by this work, obtaining better results than the considered ones. It will be available *a posteriori* online on GitHub[2].

The DSRM process model provides for the communication and sharing of the results obtained (see step *6. Communication* in image 1.1). This communication will be done in two main ways:

- The dissertation, due in October 2024, which will be available *a posteriori* for public consultation in the University of Beira Interior (UBI) library's digital archive;
- A scientific paper, planned for early 2025, which will summarise the findings achieved in this project.

1.5 Dissertation Organisation

This document is organised as follows:

- Chapter 1: Introduction – mentions the scope, motivation, problem statement, objectives, and approach related to this project; details the main contributions; presents the document's organisation;
- Chapter 2: Background and State-of-the-Art – an insight over the theoretical bases acquired and required for the project: system architectures, namely microservices, and anomaly detection; presents the state-of-the-art solutions for both these issues;
- Chapter 3: Implementation Overview – the dataset utilised, also detailing the transformation to graphs, training, and testing, and overviews GNNs and performance metrics;
- Chapter 4: Results – the testing details, obtained results and the respective discussion,

[2] `https://github.com/pipademus/`

and the answers to the research questions;

- Chapter 5: Conclusion – main conclusions to be retrieved from the project, threats to its validity that should be considered, as well as future work to be held.

Chapter 2

Background and State-of-the-Art

2.1 Introduction

Before going into the details of the implementation of the algorithm, obtaining the theoretical bases that support it is a must. Besides defining what microservices architecture is, knowing the programmatic bases to be used in its implementation is also a step of major importance.

This chapter is therefore divided into three sections:

- Section 2.2 – presents the monolith, service-oriented, and microservices architectures, and compares them;
- Section 2.3 – defines what are anomalies and presents some common aspects of detection systems;
- Section 2.4 – presents examples of the application of the microservice architecture, and discusses solutions for detecting anomalies in microservices.

2.2 Architectural Overview

Although there is not a common definition for what is a software architecture, there are some guidelines about what a software architecture should provide. Architectures should communicate about the software they represent in three main points [27]:

1. Explaining the software's structure;
2. Guiding over the common accepted rules;
3. Helping implement the system's requirements.

Thus, Jaakkola and Thalheim propose the following definition: *a system architecture represents the conceptual model of a system, including structural, behavioural and collaboration elements* [27].

2.2.1 Monolithic Architecture

The basic unit of monolithic architecture is the monolith. Sam Newman defines monoliths as a single unit of deployment where all functionalities of the system must be deployed together. Monoliths can be categorised in three different ways [7]:

- Single-Process Monolith – all the code is considered as a single process;
- Modular Monolith – the single process consists of separate modules, which can be worked on independently, but must be deployed combined;
- Distributed Monolith – the single process is composed of multiple processes, but the entire system must be deployed together.

All logic for executing a request runs a single process, which can be run and tested on a developer's laptop. However, every single change made requires the entire monolith to be rebuilt and deployed, leading to numerous versions. The scalability of a monolith is achieved horizontally by running as many instances of the system as necessary behind a load balancer [2].

The simplicity of the monolithic architecture is its main advantage, allowing for simpler development workflows, monitoring, and troubleshooting [7]. The monolithic architecture is recommended for start-up systems because it helps to explore both the complexity of a system and its component boundaries [5].

2.2.2 Service-Oriented Architecture

The Service-Oriented Architecture (SOA) is defined by Sam Newman as a design approach in which various services work together to deliver a specific end set of capacities. In the SOA, services are a fully separated process which communicate via external calls through a network rather than via internal method calls [7].

The International Business Machines Corporation (IBM) states that SOA specifies how to make software components reusable and interoperable. Services are made available via common network protocols and can quickly be integrated into new applications by using an architectural pattern and common interface standards. Each service contains all data and code needed to carry out a single, unique business function [28].

The SOA intends to promote the reusability of software by making it easier to maintain and rewrite. In theory, services must be able to be altered, replaced, or even eliminated without it being noticed by the end user [7]. There are three major advantages in the SOA [28]:

- Greater Business Agility – reusable services allow for faster assembly and deployment of applications;

- Leverage Functionality – proprietary functionalities can be extended and explored in different platforms, markets, or environments;

- Improved Collaboration – services can be defined in business terms, allowing for easier communications within organisations.

2.2.3 Microservice Architecture

Sam Newman describes microservices as services modelled according to a business domain which can be released independently. Each service encapsulates a unique functionality, which internal implementation details are hidden, and delivers it via networks [7].

Fowler and Lewis define microservices as an approach to developing a single application as an ensemble of services, each running a unique, specific function, which communicate with lightweight mechanisms. Every service can be built, deployed, and managed individually, with different programming and data storage languages and technologies, requiring limited centralised management [2].

Microservices follow the concept of information hiding: the information is exposed as little as possible through external interfaces. This allows for a clear separation between the services within a system, thus, changes within a microservice do not affect other microservices [7].

There are some key aspects one must consider when studying microservices [2, 7]:

- Design for Failure – any service can fail at any time, so the application must be designed to react quickly and smoothly to any abnormal occurrence. To be able to detect the failures in the smallest time range and, if possible, to automatically restore the service's functionality is of major importance;

- Domain-Driven Design – each microservice is built to perform a single, discrete functionality concerning the real-world environment of the system;

- Flexibility – organisational, technical, scaling, and robustness choices are particular to each microservice in a system, depending on its functionality;

- Independent Deployability – each microservice can be changed, tested, and deployed without the need to intervene on other microservices, maintaining explicit, well-defined, and stable communication among them;

- Organisation Alignment – grouping developers in multi-skilled teams in order to implement back-end (server-side logic), front-end (user interface), and database for each microservice in a fast, agile way. Cross-functional teams have the full range of skills necessary for the development of coherent, optimised services;

- Ownership – each microservice has the authority to decide what is shareable with other microservices and what is private, encapsulating data and behaviour within. This allows for loose coupling between the parts of a system, as well as for closer relationships between customers and developers to provide better products;

- Size – a microservice should be kept at the size where it is easily understood, both by developers with extensive knowledge of the system and by recent developers.

Because microservices allow for incremental modernisation, systems with high availability and scalability can be created at lower costs and with redundancy of service instances. In addition, microservices must be able to overcome three main challenges [6]:

- Multi-Tenancy – the capacity to meet the needs of several service consumer groups, organisations, and even challengers, allowing companies to share access to the same physical and application instances while maintaining strict data separation and control;

- Statefulness – the capacity to preserve previously generated state information. Ideal microservices are stateless to better exploit the benefits of cloud computing, namely load balancing, on-demand elasticity, high availability, and high reliability through redundancy;

- Data Consistency – the capacity to alter data stored in a shared repository only in permitted ways, allowing for numerous instances to be run concurrently.

Each service can be built, tested, and deployed independently; thus, changes can be rapidly delivered from development to production. This high frequency of releases means that fine-grained monitoring techniques supplement, or even replace, service quality assurance. Performance-relevant data in microservices is collected from the microservices themselves and their dependencies, and changes with almost every modification to the services. These characteristics make it extremely difficult to define a normal operation of microservice systems [29].

Despite all the advantages, starting a new project directly leveraging microservices is to be avoided. Microservices do not suit all applications, mostly due to the inherent complexities and subtleties of distributed and service-oriented computing, but also due to the fine-grained, highly flexible, and dynamic nature of microservice architectures [4, 5].

2.2.4 Architectural Comparison

As noted in section 2.2.1, in monoliths, all logic for handling a request runs in a single process. This allows for easy testing, both at the local machine and at the deployment pipeline. The testing process is not so linear in microservices, requiring the distribution of the test suite and the verification of multiple endpoints per request [2, 7].

Every change, either big or small, in the monolith's code requires the deployment of the entire

application. In microservices, changes only require the deployment of the modified services, with no need to put all the system down for the update [2].

The scaling of applications is also a point of divergence between monolithic and microservice architectures. The schematic of the horizontal scaling[1] of a monolithic application can be seen at the left side of figure 2.1, where the scaling of the application is done by running it in four different servers. Microservices applications, instead, can be scaled according to the most used or requested services in the different servers, saving resources [7]. This can be observed at the right side of figure 2.1, where each service is scaled in the different servers depending on the demand.

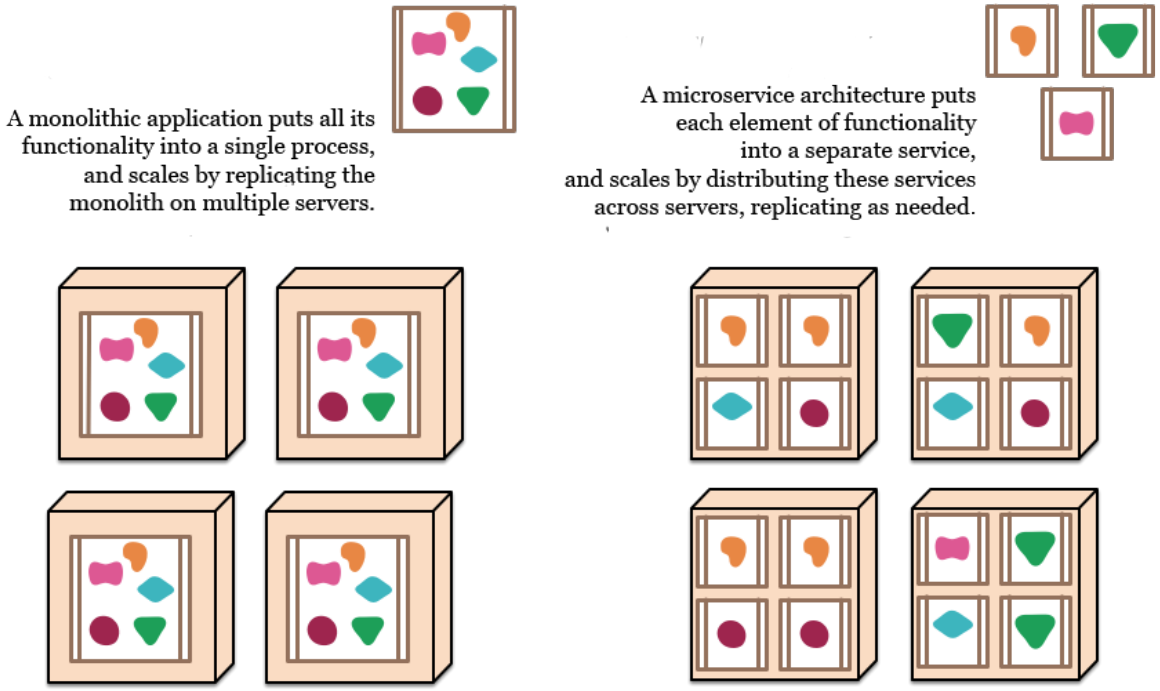

Figure 2.1: The difference in scaling monolithic applications (left) and microservices applications (right) (redrawn from [2]).

There is no consensus in the scientific community on the relationship between SOA and microservices:

- Sam Newman, Olaf Zimmermann, and Tetiana Yarygina support microservices not as a new, different architectural style compared to SOA, but an extremely specific implementation of SOA. The theoretical bases are very similar, only the execution differs [4, 7, 31];

- Robert Heinrich argues that microservices are an advancement of SOA [29];

[1]Horizontal scaling refers to increasing the capacity of a system by adding additional machines. Vertical scaling refers to increasing the capacity of a system by adding capability to the machines already being used [30].

- IBM states that, whereas SOA is an integration architectural style which enables applications to be exposed over loosely-coupled interfaces at an enterprise level, microservices are an application architectural style that enables applications to be separated into small pieces that can be changed, scaled, and managed independently [28];

- Fowler and Lewis defend that, because SOA can have a broad set of definitions, and even though microservices can be seen as a subset of that architectural style, it is beneficial to have a unique, separate definition for the microservice architectural style [2].

Table 2.1 shows the comparison between the three architectures studied: monolithic, SOA, and microservice. The table considers the previously discussed deployment, granularity, reusability, scaling, and testing features [2, 5, 7, 28, 29].

Table 2.1: Comparison between the three studied architectures: monolithic architecture, SOA, and microservices architecture.

Feature	**Monolithic**	**SOA**	**Microservices**
Deployment	Single unit	Multiple services	Independent services
Granularity	Single, large application	Larger, loosely coupled services	Smaller, independent services
Reusability	Limited	Moderate	High
Scaling	Limited, preferably vertical	Moderate	High, horizontal
Testing	Traditional, end-to-end	Individual, service virtualisation	Decentralised, chaos engineering

2.3 Anomaly Detection

Anomalies, also known as faults or failures, refer to the inability of a device or service to function correctly. Failures have external manifestations, called symptoms, which serve as signals that a component has had its behaviour deviated from the normal, expected operation [11].

Anomalies in microservices can be of multiple types [32]:

- Configuration – modifications in settings, deployment, or environments;

- Dependency:

  - Changes in the infrastructure or service communication patterns;

  - Missing dependencies;

- Error:

  - Increase in service failures or issues;
  - Frequent or repetitive error patterns;
- Outage:
  - Unexpected and/or prolonged service disruptions;
  - Avalanche failures through the system;
- Performance:
  - Unusual response times or latency;
  - Sudden spikes or drops in resource utilisation beyond normal;
- Resource Leak Anomalies – memory or resource leaks that lead to degrading performance and unnecessary use of resources;
- Scalability – over or under-provisioning of resources;
- Security:
  - Unexpected or unauthorised accesses;
  - Unusual network traffic;
  - Unusual user behaviour.

Fault management systems are developed to detect, identify, locate, and recover any failure or anomaly that degrades network or computational systems performance. Their complexity increases in proportion to the size, unreliability, and non-determinism of the systems under analysis. Fault management systems are usually composed of two parts [11]:

- Fault Diagnosis – comprising fault detection, fault localisation, and testing. The system observes the network's execution, and scans for any possible failure. The symptoms found are analysed and processed to discover the exact location of the failures, which are then tested to verify the anomaly;
- Fault Recovery – repair the fault and the respective node or service after their identification.

As discussed in section 2.2.3, a microservice system is, by nature, very complex and dynamic. This is mostly due to the extremely fine-grained and complicated interactions between the microservices and the complex configurations of the runtime environments [23, 33].

According to the scaling requirements, microservice instances may be dynamically created or destroyed. This dynamic, coupled with the frequent deployments within a microservice system, continuously changes the dependencies and behaviours of its services [24, 33].

In microservices, a faulty service affects other services, and in extreme cases produces an avalanche effect that culminates in the failure of the entire system [23, 25, 34]. To improve the reliability and robustness of microservices, detecting faults and their root causes as soon as possible in an efficient and accurate manner is therefore of major importance [12, 25].

Traditional fault detection and localisation approaches do not support the multi-dimensional nature of microservices. To monitor a system's execution, not only are the execution paths of a process relevant, but also the microservice instances, call chains, and environmental configurations [9].

Some anomaly detection approaches manually select the monitored metrics of hardware, middleware, and software, and set alarm rules triggered by the symptoms based on correlations between them. Other approaches collect execution traces to record the sequences of processing requests, requiring experts with domain knowledge to survey niche applications. Some approaches detect performance anomalies by detecting the deviation of execution times for each service, which oblige the developers and analysts to consider different possible times according to different execution paths [25].

Approaches for anomaly detection in microservices can be divided into three main categories [23, 35]:

- Log Based – identify the faulty components based on parsed system logs. Advised to find informational causes, they are hard to work in real time and require abnormal information to be hidden in the logs;

- Metric Based – construct the causality graph among components by collecting the metrics from application and infrastructure data. Guaranteeing an accurate causality graph is the major challenge;

- Trace Based – gather information by examining the execution paths, and detect potential anomalies via outlier analysis. It requires significant domain knowledge and system logic.

2.4 Related Work

Theory helps understand the bases, but practical examples can prove its usefulness. As such, the following subsections analyse some of the most pressing examples on both microservices and anomaly detection: subsection 2.4.1 presents some examples of companies utilising microservices, and subsections 2.4.2 and 2.4.3 focus on anomaly detection methodologies.

2.4.1 Microservices in Industry

Many companies have transitioned from monolithic architectures to microservices, either as a necessity or as a way to modernise their infrastructure. Some of them make their microservices' implementation public, allowing for its study. A few examples are provided below.

Amazon: In 2001, delays and challenges in the development, coupled with the complex service interdependencies, made it difficult for the company to scale up and meet the demands of its exponential growth. Thus, Amazon broke its monolithic infrastructure to implement services: small, independent, service-specific applications [36].

Prime Video, one of Amazon's services, had a quality monitor feature which was not designed to endure high scale. Its execution became too expensive and began presenting many bottlenecks, which led the team to reevaluate the infrastructure and to refactor it to an architecture closer to the monolith [37].

eBay: eBay started introducing microservices in 2011. By dividing databases, application, and the search engine itself in microservices, they were able to face the challenges of system complexity, developer's productivity, and faster time-to-market [38].

Etsy: Etsy transitioned to the microservice architecture in 2016 after experiencing performance issues caused by sequential processing bottlenecks and needing to expand to new features and platforms. The desired Application Programming Interface (API) concurrency was achieved via cURL[2] for parallel execution [36].

Microsoft: When migrating their standalone legacy apps into discrete services in 2011, Microsoft developed Service Fabric[3], which powers most of their internal and external services and is available for common use. Service Fabric supports stateful and stateless microservices as well as other utilities, such as life-cycle management, hybrid deployments, and high availability [40].

Netflix: Netflix started their migration to the cloud in August of 2008, after experiencing a severe case of database corruption that immobilised their functioning for three days. The migration was concluded in early 2016. One of the biggest changes was their system architecture, with the enterprise switching from a monolithic application to hundreds of microservices [41].

Spotify: Spotify uses microservices as a way to focus on autonomous full-stack teams in the development and execution of its business objectives, to facilitate asynchronicity. Spotify open-sourced their microservices' baseline, called Apollo, in GitHub to allow for others to develop their own systems with their framework [42, 43].

[2] cURL is a command line tool and library for transferring data with Uniform Resource Locators (URLs) [39].
[3] `https://azure.microsoft.com/en-us/products/service-fabric/`

Uber: Uber Technologies, Inc., started their migration to a microservice architecture in 2012-2013, when their two monolithic services started showing some problems regarding availability, deployment, separation of concerns, and execution. Uber then adapted their microservices approach and designed the Domain-Oriented Microservice Architecture (DOMA) [44].

2.4.2 TRL9 Approaches in Anomaly Detection

Technology Readiness Level (TRL) is a method for evaluating a technology's maturity which allows for consistent and standardised discussions of technological maturity across distinct fields [45]. In TRL 9, the real system is already implemented in an operational production environment.

Even though most companies do not disclose their internal practices in anomaly detection due to proprietary reasons, some of them make them public, thus allowing for them to be analysed.

Amazon: Although Prime Video's service architecture was recently switched from microservices to a monolith, its anomaly detection process while still in the microservice architecture was partly disclosed. The algorithm considers distinct categories of deviations provoked by varying customer behaviours in order to diminish the triggering of false alarms. This differentiation is particularly difficult since benign dips in metric traffic might resemble those resulting from real-world incidents. By considering external variables in the anomaly detection process, predictions and monitoring activities can be cross-validated to provide the most accurate results [46].

eBay: eBay developed Moving Metric Detector (MMD), an algorithm which commences by decomposing time series in normal and abnormal samples, from which a list of time series with potential anomalies is gathered. Then, a more refined framework is applied in order to improve the accuracy and precision of reporting true anomalies, and to deliver a final list of possible anomalies that must be investigated [47]. Another system leveraged by eBay is Graph-Based Approach of Microservice Trace Analysis (GMTA), which processes traces in real time to detect anomalies and localise root causes [24].

Etsy: Etsy created its own open-source alerting framework, `411`, over Elastic Stack, in 2014. It uses stored search queries as a reactive security mechanism, which allows for customisable alerts based on user queries to improve the reaction to important security events. It also permits integration with other alert and monitoring systems, while keeping a registry of relevant events [48].

Microsoft: Microsoft required an anomaly detection system which provided solutions for the three main problems faced by the state-of-the-art systems: lack of labels, generalisation, and efficiency. Thus, they developed an algorithm using the Spectral Residual (SR) method

(unsupervised learning) and Convolutional Neural Network (CNN) (supervised learning). SR has a great performance in detecting visual saliencies, and is directly applied on the raw time-series data. The CNN's input is the SR's output, which can be assumed as labelled, and thus achieves the supervised learning results without the cumbersome task of primarily labelling data points [49].

Netflix: In 2022, Netflix implemented an anomaly detection approach leveraging both rule-based and model-based methods. It encompasses both semi-supervised and supervised model-based approaches. Data is extracted from streams and stream-related sources, from which the most important features are obtained to train the anomaly detection models. These models are semi-supervised when learning the distribution of data samples in training or validation datasets, and supervised when classifying them [50].

Spotify: Spotify leverages a sophisticated anomaly detection system to spot and neutralise anomalous occurrences, comprised of four main steps: data collection, preprocessing, anomaly detection, and action. Its algorithms are always being polished, with constant optimisations of the Machine Learning (ML) and Artificial Intelligence (AI) parts to ensure the models are up to date with the possibly malignant technologies [51].

Uber: Uber launched, in 2020, Data Quality Monitor (DQM) as a solution which utilises statistical modelling to connect the various components of data quality analysis, based on historical data patterns. DQM verifies whether current data deviates from the expected observations, characterising not only data quality but also high-level changes [52].

For each data table, an in-house metric generator produces a multi-dimensional time series output, which is considered as bundles. Then, the evolution of the metrics is decomposed using Principal Component Analysis (PCA) into a few representative sets for more scalable computation. These are used afterwards to obtain multiple principal component time series which represent the entire data table, including changing and seasonal trends. The last stage is one-step ahead forecasting, meaning predicting only the next value in a time series to check whether it follows past trends. If it does not, it can be considered an outlier [52].

2.4.3 Research Approaches in Anomaly Detection

Kitchenham proposes a research methodology consisting of three steps, the Systematic Literature Review (SLR), which can be seen in figure 2.2.

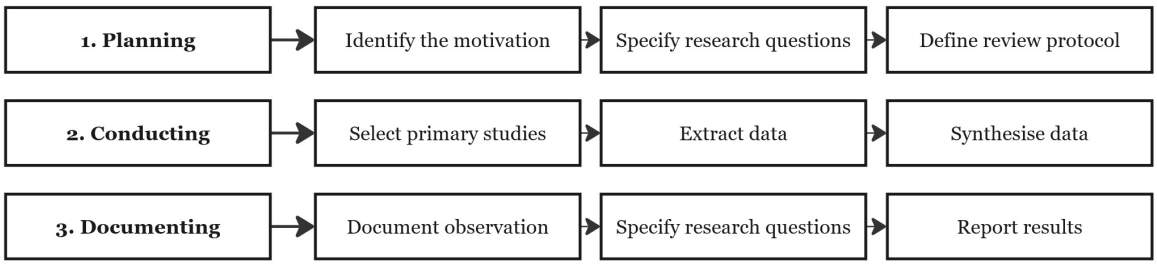

Figure 2.2: The SLR research methodology (adapted from [3]).

This methodology was followed when conducting the state-of-the-art study for this project, to formalise the process of conducting the search. The three steps of the SLR are defined as follows [53]:

1. Planning:

    - Identify the Motivation – detail the motives behind the project, described in section 1.1;

    - Specify Research Questions – propose the questions to be answered by the work, detailed in section 1.2;

    - Define Review Protocol – determine the problem's scope, to adjust the search terms, overviewed in section 1.1;

2. Conducting:

    - Select Primary Studies – designate the beginning points for the literature review. First, the paper by Meng et al. was used to start cross-referencing with other papers [25]. Then, to enlarge the research scope, the IEEE database was searched with the following keywords: `microservices anomaly detection`. The inclusion and exclusion criteria are defined in table 2.2;

    - Extract Data – gather the important data from the final list of papers;

    - Synthesise Data – cross-check data from various sources to obtain a robust summary of the information;

3. Documenting:

    - Document Observation – write a report with the information gathered in step `2. Conducting`;

    - Analyse threats – extrapolate potential threats to the validity of the information, as done in section 5.2;

- Report Results – publish and/or share the final report.

Table 2.2: The inclusion and exclusion criteria for the SLR.

Inclusion	Papers presenting solutions on anomaly detection in microservices systems
	Papers more recent than 2018
	Papers mentioning the migration from monoliths to microservices
Exclusion	Papers which mention anomaly detection, but not microservices
	Books or book chapters, excluding [7]
	Papers which do not present their results in terms of accuracy, f-score, precision, or recall

The final list of selected papers is presented below. Do notice that there might be papers which also concern anomaly detection in microservices systems which are not studied here.

These are the sixteen approaches considered in this dissertation:

- Arsalan et al. developed Statistical Learning-Based Metric Anomaly Detection (SLMAD), a three-part framework which analyses time series, dynamically groups them, and trains and evaluates models on that data. SLMAD uses box-plots, a graphical demonstration through quartiles, and matrices [13];

- Jinjin et al. propose MicroScope, a system composed of three parts: data collection, which gathers network information and metrics; service causality graph building, which constructs a Directed Acyclic Graph (DAG) based on the relationships and dependencies from the data; and cause inference, which traverses the graphs to obtain a list of potential anomalies [54];

- Li et al. present an algorithm which applies trace data to train low-value, mutation, and anomalous features, creating clusters of normal and abnormal data. Then, a dynamic sliding time window is applied to trigger the anomaly detection process [20];

- Li et al. introduce TraceRCA, a three-fold approach which detects trace anomalies, mines suspicious microservices sets, and ranks them. TraceRCA uses hard-coded rules, coupled with a frequent pattern mining technique called FP-growth and the computation of Jaccard's Index, a similarity statistic metric [15];

- Meng et al. propose a three-part approach that collects execution traces across microservices and describes them via call trees, calculates the anomaly degree of traces through tree edit distance to locate suspicious components, and checks response time fluctuations with PCA to detect anomalous calls [25];

- Jing et al. developed Adaptive Label Screening and Relearning (ALSR), that uses a label screening model to remove unnecessary data from the training set and a relearning model to analyse continuous intervals of Key Performance Indicators (KPIs). Important features are extracted and processed by a Deep Neural Network (DNN), and the

data is processed by a Random Forest (RF) algorithm [55];

- Wang et al. introduce iRRCF-Active, an anomaly detection which contains an unsupervised and white-box anomaly detector using Robust Random Cut Forest (RRCF) and active learning. Active learning is incorporated by user feedback, providing the model with high-quality identifiers provided by specialists [56];

- Zeng et al. present TraceArk, an anomaly evaluator that aggregates time series data for call chains and metadata for easier data retrieval. Alerting windows are applied to reduce noise, and anomaly scores are calculated for each instance. Engineers label anomalies, which are used to update anomaly thresholds via semi-supervised learning with an XGBoost model [19];

- Zhou et al. developed Microservice Error Prediction and Fault Localisation (MEPFL), based on learning from system trace logs. Based on a set of features, MEPFL trains prediction models at trace and microservice levels with normal and abnormal data. These models, using RF, K-Nearest Neighbours (KNN), and Multi-Layer Perceptron (MLP) algorithms, are then applied in production environments to predict errors and faults [12];

- Chen et al. propose the use of a Request Weighted Directed Graph (RWDG) and a Request String (RS) to model the procedure of request processing, diagnosing possible anomalies with a DNN model. The algorithm collects the call information of microservices in terms of response times, characterising them with weights to build a directed weighted graph. The model is maintained using the Adam algorithm for the feed-forward transfer process [14];

- Chen et al. present TraceGra, an unsupervised encoder-decoder approach divided in three stages. First, it constructs a unified graph representation of traces and metrics. Second, it leverages a Variational Graph Auto-Encoder (VGAE) and a Long Short-Term Memory (LSTM) with Auto-Encoder (AE) to obtain topological and temporal features. Third, it calculates the anomaly score by adding the two-part loss value with two hyperparameters [18];

- Lee et al. introduce Eadro, an end-to-end framework integrating anomaly detection and Root Cause Analysis (RCA) based on multi-source data. Eadro models the internal and external behaviours of the services from traces, logs, and KPIs with Hawkes process to dense vectors. Graph Neural Networks (GNNs), like Graph Attention Network (GAT), are then applied for triage, dynamically assigning weights to nodes for the anomaly detection [21];

- Ping et al. propose TraceAnomaly, where service traces are used to generate models via deep Bayesian networks which are periodically retrained. Call paths and response

times are extracted from traces, forming service trace vectors that allow for unified patterns. The anomaly detection algorithm applies a Variational Auto-Encoder (VAE), trained through Stochastic Gradient Variational Bayes (SGVB) and evaluated using log-likelihood against the model [57];

- Scheinert et al. developed TELESTO, a system combining topology-adaptive Graph Convolutional Neural Networks (GCNNs) and GATs to generate meaningful node representations. These layers can be stacked randomly. It also incorporates a final feed-forward block, as well as batch normalisation, one-dimensional convolution, and a softmax layer for computing class probabilities [16];

- Zhang et al. present DeepTraLog, which uses a unified graph representation to typify traces and logs within a system. DeepTraLog trains a Gated Graph Neural Network (GGNN) based on deep Support Vector Data Description (SVDD) models. The anomaly detection process feeds a trace event graph to the model to generate a latent representation, where it calculates the anomaly score [17];

- Zhe et al. propose TraceVAE, an unsupervised anomaly detection using Graph Variational Auto-Encoders (GVAE). TraceVAE also proposes a novel dispatching layer through the decomposition of negative log-likelihood into Kullback-Leibler divergence and data entropy, to permit anomalous scoring in anomaly detection processes [58].

The research approaches analysed previously have their main results compiled in table 2.3, ordered by year of publication and name of the first author.

2.5 Conclusion

The microservice architecture is an advantageous approach which can help companies achieve their goals more easily. However, this requires developers and stakeholders to thoroughly consider whether this is the best option to be applied. As noticed in Amazon's example, sometimes the most traditional monolithic architecture is the way to go. Companies must take into account that there is not a one-size-fits-all solution.

Still, the microservice architecture is widely adopted, and it requires a particular, novel way to be monitored. Academia and industry continue to strive to find the most lightweight, effective, and efficient solutions to detect and repair anomalies as quickly as possible. The developed approaches leverage rule-based, ML, and Deep Learning (DL) algorithms in their solutions to unburden the developer's need to hard-code rules and to optimise hyper-parameters, as well as to speed up adaption to code changes.

Having this *a priori* contextualisation, to advance to the development of a new algorithm expected to compete with the presented state-of-the-art solutions is now possible.

Table 2.3: Summary of the state-of-the-art considered for this dissertation on anomaly detection, ordered by year and name of the first author. A: Accuracy, F: F1-Score, P: Precision, R: Recall.

Year	Works	Dataset	Method	Split Ratio	Performance
2018	Jinjin et al. [54]	Sock Shop	DAG	Unspecified	P: 88.0% R: 85.0%
2019	Jing et al. [55]	AIOps 2020	DNN, RF	70-0-30	F: 96.5%
	Zhou et al. [12]	TrainTicket Sock Shop	KNN, RF, MLP	Unspecified	F: 92.7% P: 98.6% R: 87.4%
2020	Arsalan et al. [13]	Huawei	Box-plots and matrices	70-0-30	F: 64.6% P: 85.5% R: 56.5%
	Ping et al. [57]	TrainTicket	VAE, SGVB	Unspecified	P: 98.0% R: 97.0%
	Wang et al. [56]	AIOps 2020	RRCF	Unspecified	F: 89.0%
2021	Chen et al. [14]	TrainTicket	RWDG, RS, DNN	Unspecified	A: 91.5% F: 88.6% R: 89.0%
	Li et al. [20]	AIOps 2020	Hard-coded rules	Unspecified	A: 98.3%
	Li et al. [15]	TraceRCA TrainTicket	Hard-coded rules	20-0-80	F: 80.0% P: 85.0% R: 80.0%
	Meng et al. [25]	Bench4Q Social Net-work	Call trees, PCA	Unspecified	P: 89.8% R: 88.3%
	Scheinert et al. [16]	Generated	GCNN, GAT, LSTM	Unspecified	A: 85.1% F: 85.4% P: 86.1% R: 85.1%
2022	Zhang et al. [17]	TrainTicket	SVDD, GGNN	60-10-30	F: 95.4% P: 93.0% R: 97.8%
2023	Chen et al. [18]	TraceRCA, TrainTicket	VGAE, LSTM-AE	Unspecified	F: 95.3% P: 97.1% R: 93.5%
	Lee et al. [21]	TrainTicket Social Net-work	GAT	60-0-40	F: 98.9% P: 98.4% R: 99.5%
	Zeng et al. [19]	Exchange TrainTicket	XGBoost, hard-coded	Unspecified	F: 95.3% P: 95.0% R: 95.8%
	Zhe et al. [58]	Proprietary	GVAE	Unspecified	F: 91.8%

Chapter 3

Implementation Overview

3.1 Introduction

After obtaining the necessary theoretical bases, as well as studying the previous works in the field, one must understand the implementation details to create a potential solution. This chapter is thus divided in four main parts:

- Section 3.2 – presents the dataset leveraged in this project, detailing its characteristics and features;
- Section 3.3 – conceptualises de model and details the transformation of the data to graphs;
- Section 3.4 – discusses some theoretical bases on neural networks, particularly, Graph Neural Networks (GNNs). In this section, more practical details are shown, such as the type of applied convolutional layers, model settings, and training and evaluation features;
- Section 3.5 – presents the metrics and values used to assess the performance of the model.

3.2 Dataset

A dataset is a group of connected, discrete records representative of a domain [59]. Each record corresponds to an observation of the field.

Traces are representations of the entire execution process of a user request, comprised of spans connected amongst themselves in parent-child relationships. A span is a unit of work within a distributed system, characterised with attributes which show the overall status of each component, like the identifier, the parent span identifier, and the trace identifier [60].

Traces and spans can be represented with graphs[1], which can then be fed into Machine Learning (ML) models for various purposes.

[1] A graph G is a pair $G = (V, E)$, where V is a finite set, called the vertices of G, and E is a subset of $\mathcal{P}_2 V$ (the set of all 2-element subsets of the V set), called the edges of G [61].

3.2.1 Characterisation of the Dataset

The AIOps Challenge 2020 is a dataset consisting of 19.303.636 records of trace data, each representing a span. This dataset was chosen not only due to the great amount of information, but especially because the data was collected within an internal Chinese wireless provider system and thus represents a real-world workload [60]. However, there is little literature concerning the application of this dataset in just the anomaly detection, as its original purpose is checking how to develop better Root Cause Analysis (RCA) methods.

The AIOps Challenge 2020 dataset is available publicly on GitHub [62]. The dataset is composed of fifteen folders concerning different days of logging data, as well as an extra folder with auxiliary files. From these, only the data referring to the 31st of May 2020 was chosen for training and testing purposes. Each date folder contains three subfolders:

- Business Indicators;
- Platform Indicators;
- Call Chain Metrics – the one used in this project. It contains six data files:
  - `trace_csf.csv`;
  - `trace_fly_remote.csv`;
  - `trace_jdbc.csv`;
  - `trace_local.csv`;
  - `trace_osb.csv`;
  - `trace_remote_process.csv`.

These six `.csv` files contain the data which is going to be fed to the model to train, validate, and test it.

3.2.2 Characterisation of the Features

The six data files mentioned in section 3.2.1 represent data under ten features. These are, by order of appearance:

- `callType` – execution type: `CSF`, `FlyRemote`, `JDBC`, `Local`, `OSB`, `RemoteProcess`;
- `startTime` – timestamp of the start of the span, in milliseconds;
- `elapsedTime` – time of execution of the span;

- `success` – result of the call: `True` if successful, `False` otherwise;
- `traceId` – unique identifier of the trace;
- `id` – unique identifier of the span;
- `pid` – unique identifier of the parent span;
- `cmdb_id` – unique identifier of the network element where the service runs;
- `serviceName` – name of the executing service of the span. In the `trace_jdbc.csv` file, this field does not exist;
- `dsName` – data source of the service of the span. This field only exists in the `trace_jdbc.csv` and `trace_local.csv` files.

3.2.3 Feature Inclusion/Exclusion Criteria

From the ten original features provided in the AIOps Challenge 2020 dataset, seven are utilised to create the graphs from the data. These are, by order of usage:

- `startTime` – orders the spans from the six files presented in subsection 3.2.1 in chronological order;
- `traceId` – marks the beginning of a new, unique graph;
- `id` – denotes a single node in the graph;
- `pid` – creates a relationship in the call chain between the current span and the previous span, represented by an edge in the graph;
- `elapsedTime`, `serviceName` or `dsName`, `success` – attributes used by the model to learn and evaluate the anomaly detection process.

These features, and in particular, `elapsedTime`, `serviceName` or `dsName`, and `success`, were chosen due to being easier to generalise for other datasets. This allows for a more simple, generic approach, which is one of the problems this work intends to solve.

3.3 Conceptualisation

Before implementing the model and experimenting with it, one needs to conceptualise the idea. This can be done, for example, like is shown in figure 3.1. The data mentioned in the first block comprises all the records of the AIOps Challenge 2020 dataset, reviewed in section 3.2.

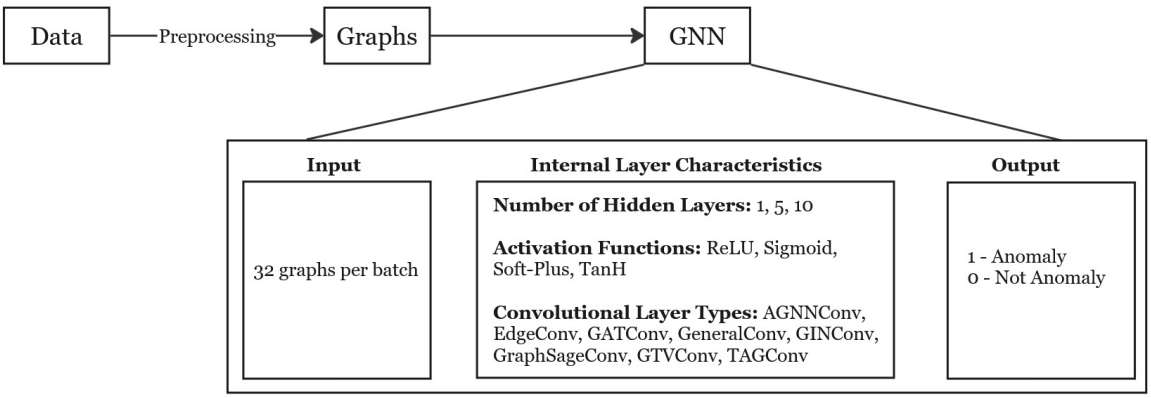

Figure 3.1: The conceptual model of the algorithm.

In order to use GNNs, the traces from the AIOps Challenge 2020 dataset have to be converted into graphs. This process is described in algorithm 1.

The algorithm first reads and collects spans from multiple data files, storing them in a list. It then sorts the spans and, using the ordered list, retrieves features for each span. Based on these features, it builds a node-feature matrix, adjacency matrix, and a node-label list to create a graph. `Spektral` has a built-in functionality which allows to convert matrices into graphs, which is used on this step. Finally, it returns the list of generated graphs.

Algorithm 1 Pseudo-code for the preprocessing of the data.

```
1:  ## get all the spans and order them ##
2:  for file in span_files do
3:      read data file
4:      append spans to list
5:  sort span list
6:  ## create graphs from the ordered span list ##
7:  for span in span_list do
8:      retrieve features
9:      build node-feature matrix, adjacency matrix, node-label list
10:     create a graph
11: return graphs
```

Some extra information about algorithm 1:

- In line 5, the list is sorted by `traceId` and `startTime`;
- In line 8, the retrieved features are: `id`, `pid`, `elapsedTime`, `serviceName` or `dsName`, and `success`;
- In line 9, the three created structures are defined as follows:

- `node-feature matrix` – a matrix with three columns, one for each feature, which values are encoded;
- `adjacency matrix` – the adjacency matrix of the graph;
- `node-label list` – a three-slot list which contains the training values for the learning of the model.

Do note that this project does not leverage normalisation, as the data can be processed by the neural network model without modifications [63]. Over-sampling techniques, which allow for a better balance of the input data, like SMOTE, were not applied [64].

3.4 Graph Neural Networks

A neural network is an ML algorithm which uses methods that imitate how biological neurons collaborate to recognise occurrences, evaluate possibilities, and draw conclusions. Neural networks are composed of an input layer and an output layer (visible layers), with layers between them (hidden layers). They are effective tools in Artificial Intelligence (AI) which allow for a quick classification and clustering of data [65].

In an ideal execution, the predicted values of a neural network $\hat{y}$ match the expected values y, implying a loss value of 0. Usually, the predicted values $\hat{y}$ are interpreted as probabilities [66].

In the training phase, a neural network receives input values x and the corresponding outputs y. When processing a given input vector, the neural network executes several intermediate operations before returning the output. The values calculated at each layer of the network can be seen as different representations of the given input, transformed from the representation produced by the preceding layer [66].

In classification problems, the neural network outputs a categorical distribution, i.e., if there are d possible answers, there are d output nodes which probabilities sum to 1. This is achieved by applying an output layer which returns a vector of d values given a vector of input values [66].

GNNs are neural networks designed specifically to operate on graph-structured data. Unlike traditional neural networks, GNNs can accurately capture the relationships and interactions between nodes in a graph [67].

The core idea of GNNs is to iteratively update the representation of each node by aggregating feature information from its neighbour nodes. This process allows the network to learn features within the structure of the graph and the positioning of the nodes, which makes GNNs rather powerful in various contexts, like anomaly detection in microservices [67].

3.4.1 Activation Functions Definition

In each hidden layer, an activation function introduces non-linear complexities to the model to develop better representations of the data. These functions include Hyperbolic Tangent (TanH), Rectified Linear Unit (ReLU), Sigmoid, and Soft-Plus [68].

Considering:

- x as a real number ($x \in \mathbb{R}$),
- $f(x)$ as a real-valued function from $\mathbb{R}$ to $\mathbb{R}$,
- e as the base of the exponential function with the Euler's number, approximately equal to 2.718,
- max as the maximum value between two real numbers,
- ln as the natural logarithm function,

then the activation functions can be defined as follows [68]:

- TanH:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}; \tag{3.1}$$

- ReLU:

$$f(x) = \max(0, x); \tag{3.2}$$

- Sigmoid:

$$f(x) = \frac{1}{1 + e^{-x}}; \tag{3.3}$$

- Soft-Plus:

$$f(x) = \ln(1 + ex). \tag{3.4}$$

3.4.2 Convolutional Layers Definition

Convolutional layers are a fundamental building block of neural networks. These layers apply a convolution operation to the input data, which helps capturing spatial hierarchies and patterns within the data [67].

The `Spektral` Python library provides some examples of convolutional layers, gathered from various papers. In this project, the following are applied [26, 69]:

- `AGNNConv` – Attention-Based Graph Neural Network Layer [70];
- `EdgeConv` – Edge Convolutional Layer [71];
- `GATConv` – Graph Attention Layer [72];
- `GeneralConv` – General Layer [73];
- `GINConv` – Graph Isomorphism Network Layer [74];
- `GraphSageConv` – GraphSAGE Layer [75];
- `GTVConv` – Graph Total Variation Layer [76];
- `TAGConv` – Topology Adaptive Graph [77].

3.4.3 Model Settings

In this project, the GNN model is configured as follows:

- Number of Epochs – the network was set to train in a maximum of 400 epochs;
- Batch Size – to balance between computational efficiency and the stability of the training process, the data was divided in batches of 32 graphs;
- Learning Rate – based on common practices in the field, this value was set to 0.001;
- Early Stopping Patience – this value was set for 10 epochs: if the validation loss does not improve in that number of epochs, the training is halted to prevent over-fitting[2];
- Optimiser – the Adam optimiser, with default parameters and the aforementioned learning rate, was applied;
- Loss Function – as the anomaly detection problem concerns classification, categorical cross-entropy loss was utilised;
- Normalisation – applied to the adjacency matrix to ensure that the graph structure was properly scaled.

[2]The production of an analysis that matches a specific set of data, partially or totally, thus potentially failing to fit to new, unseen data or to predict from new values [78].

These values were chosen from an example where Spektral is applied to a custom dataset[3]. These parameters were not tuned for this project, as the focus was to find the best combination of number of hidden layers, type of activation functions, and type of convolutional layers.

3.4.4 Model Training and Evaluation

The most important part when developing any type of ML algorithm is training and testing it, to ensure the correctness of the results and to obtain the necessary information to draw conclusions. This performance data is discussed further in section 3.5.

Each epoch corresponds to a training step, where the model is trained on a batch of input data and corresponding target labels. The model processes the inputs to produce predictions and to calculate loss values. Then, gradients are computed for updating the model parameters, in order to improve the obtained results and to upgrade the model, and accuracy is computed.

The training process is detailed in algorithm 2. It processes data in batches by applying a model to the input and calculating loss and accuracy values, updating the model parameters using gradients. After analysing each batch, it evaluates the model on validation data, comparing the current loss to the best recorded validation loss. If the current loss improves, it updates the best model parameters and resets a patience counter. If not, the patience counter decreases. If patience runs out and the best loss is below a certain threshold, the training process stops.

Algorithm 2 Pseudo-code for the training of the model.

```
for batch of data do
    apply the model to the input data, with labels
    calculate loss values
    obtain the gradient values
    apply the gradients to the model parameters
    get accuracy values
    if batch fully analysed then
        evaluate model for validation purposes (algorithm 3)
        obtain loss and accuracy values
        if current loss is better than overall best validation loss then
            update best loss value
            save current parameters as best model parameters
            restart patience counter
        else
            decrease patience counter
        if patience equals zero and best validation loss is lesser than 0.01 then
            break
```

[3] https://github.com/danielegrattarola/spektral/blob/master/examples/graph_prediction/custom_dataset.py

Line 16 of the training process (algorithm 2) manages the early stopping of the model training. The two conditions guarantee two things:

- The `patience equals zero` ensures that there is no more room for improvement in the model parameters;
- The `best validation loss is lesser than 0.01` ensures the validation loss is always lesser than 1%. It must be noticed that this may not mean that the training loss is lesser than 1%, which occurs in some of the obtained results.

Evaluating the model is more complex, and it assesses its performance on the dataset. It initialises the output variables: loss and accuracy. The model iterates over the test data, computing predictions for each batch, calculating loss and accuracy, and storing the results. Then, it calculates the weighted average of the loss and accuracy over all processed batches.

The testing steps are detailed in algorithm 3. It starts by initialising variables and then iterates through batches of input data, applying the model without using labels. It calculates loss values and updates the model's parameters using gradients. When the final batch is processed, it computes the average loss and accuracy, and returns a classification report.

Algorithm 3 Pseudo-code for testing the model.

```
1: initialise output and auxiliary variables
2: while iterating through the batches do
3:     apply the model to the input data, without labels
4:     calculate loss values
5:     apply gradients to the model's parameters
6:     if final batch then
7:         calculate average loss and accuracy values
8:         return classification report
```

3.5 Performance Metrics

Evaluating the performance of neural networks is essential for understanding how well they perform when dealing with unseen data and verifying their effectiveness in solving specific problems. Evaluation metrics provide quantitative measures that help understanding the model's performance and optimising its parameters in order to achieve the best results [79][4]:

- Accuracy – presents the overall performance of the model by calculating the proportion of the correctly classified instances out of the total number of instances in the dataset:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}; \tag{3.5}$$

[4]Let it be noticed that, in all these metrics, TP stands for True Positive, TN stands for True Negative, FP stands for False Positive, and FN stands for False Negative.

- F-Score – calculates the harmonic mean of precision and recall, serving as a hybrid metric for unbalanced classes:

$$fscore = \frac{2TP}{2TP + FP + FN}; \tag{3.6}$$

- Precision – presents how accurate the positive predictions are by calculating the proportion of correctly predicted positive values out of all predicted positive values:

$$precision = \frac{TP}{TP + FP}; \tag{3.7}$$

- Recall – presents the coverage of the actual positive sample by calculating the proportion of correctly predicted positive values out of all true positive values:

$$recall = \frac{TP}{TP + FN}. \tag{3.8}$$

Performance is also measured in terms of computing resources. To check these, three additional metrics are collected during the execution of the algorithm, for both the training and testing phases:

- Central Processing Unit (CPU) consumption – calculated using a thread which verifies the CPU usage percentage every ten seconds;

- Elapsed Time – measured by subtracting the ending time with the starting time;

- Random Access Memory (RAM) Consumption – calculated using a thread which verifies the RAM usage percentage every ten seconds.

3.6 Conclusion

Implementing a model requires theoretical knowledge, but also some practical know-how. Implementing the model itself is the most challenging task, due to the specificities one must take into account and the programming limitations. By applying the functional parts and considering the insights learned when reviewing the literature, the model is more resilient to potential issues that may arise. Knowing how the different performance metrics work makes it possible to select the best to demonstrate the mathematical advantages of the model.

Chapter 4

Results

4.1 Introduction

Gathering the information compiled in chapters 2 and 3, the next step is to implement the algorithm and assess its performance, not only in terms of accuracy of its results, but also in computational resources and time invested.

To verify these, this chapter is divided in the following sections:

- Section 4.2 – details the environment and procedure specifications, including the work-around required due to the lack of codes for direct comparison;
- Section 4.3 – analyses the obtained results and draws the major conclusions;
- Section 4.4 – answers the research questions according to the achieved results.

4.2 Testing Details

To implement the algorithm and retrieve valid results, one must try to keep the training and testing environments as stable and standardised throughout the various code executions as possible. In practice, this means that the variants of the algorithm must always be run in the same computational space, which is detailed in subsection 4.2.1, and must always follow the same general steps, which are enumerated in subsection 4.2.3. Because the publicly available codes are not fully functional and the missing parts are not deducible, one algorithm was run for the baseline comparison, detailed in subsection 4.2.2.

4.2.1 Testing Environment

In its different version, the algorithm varies in the number and type of convolutional layers, as well as in the activation function type. To run the algorithm in its different variations, a computer with some more powerful specifications is required. The particular environment, in this case, is shown in table 4.1. This computer does not have any Graphics Processing Unit (GPU), which would improve greatly its performance in terms of resources.

In terms of software, the specific versions are compiled in table 4.2, for Python and its libraries.

Table 4.1: The specifications of the computational environment.

Component	Details
Central Processing Unit (CPU)	Intel®Core™i7-8700 CPU @ 3.20GHz, 6 cores, 12 threads
Operating System	Linux Ubuntu 22.04.1 x86_64
Random Access Memory (RAM)	32 GB DDR4, 2 x 16 GB modules, 2666 MT/s

Table 4.2: The specifications of the software versions.

Software	Version
`Python`	3.10.18
`numpy`	1.24.4
`pandas`	2.2.1
`pip`	22.0.2
`psutil`	5.9.8
`scikit-learn`	1.4.1.post1
`scipy`	1.12.1
`spektral`	1.3.1
`tensorflow`	2.14.1
`threadpoolctl`	3.3.0
`tqdm`	4.66.2

4.2.2 Baseline Algorithm

Some of the algorithms analysed in the state-of-the-art, in chapter 2, have their algorithms accessible online in platforms like GitHub. However, the details of how these codes work are not publicly available, either due to non-disclosure agreements or proprietary rights, and sometimes entire pieces of code or the datasets themselves are missing.

Therefore, implementing these same algorithms locally in order to have a baseline with which to compare the Graph Neural Network (GNN) developed for this project is a arduous task. Considering the time constraints faced by this project, implementing them in a timely manner was not possible. With this in mind, a variation of the tested algorithm was executed. The baseline algorithm was build without any hidden layers or activation functions to have a simple model for the comparison of results.

The results obtained for this model's execution are present in table 4.3.

4.2.3 Testing Procedure

Usually, a project has an inherent scientific method to be followed. This ensures that the results are gathered in a standardised way, and removes some variables which may appear when running multiple, similar scripts sequentially. It also allows for a more accurate way of replicating a project [80].

Table 4.3: Results obtained when using no hidden layers nor activation functions, for the baseline model.

Training	Accuracy (%)	99.398
	Loss (%)	0.561
	CPU (%)	80.652
	RAM (%)	74.152
	Elapsed Time (HH:MM:SS.mss)	02:18:26.108
	Epochs (#)	312
Testing	Accuracy (%)	99.694
	F-Score (%)	99.694
	Precision (%)	99.694
	Recall (%)	99.694
	Loss (%)	0.527
	CPU (%)	——
	RAM (%)	——
	Elapsed Time (HH:MM:SS.mss)	00:00:04.373

The *modus operandi* applied in this dissertation project is defined in algorithm 4. The convolutional layer types are defined in subsection 3.4.2, and the activation functions are specified in section 3.4.

Algorithm 4 Scientific method followed for the gathering of results.

1: **run** baseline algorithm
2: **for each** convolutional layer **do**
3: **for each** activation function **do**
4: **run** 1-layer model
5: **save** 1-layer model results
6: **run** 5-layer model
7: **if** 5-layer test accuracy **is not bigger than** 1-layer test accuracy **then**
8: **run** 5-layer model **with condition** *5-layer validation accuracy is bigger than 1-layer validation accuracy*
9: **save** 5-layer model results
10: **run** 10-layer model
11: **if** 10-layer test accuracy **is not bigger than** 5-layer test accuracy **then**
12: **run** 10-layer model **with condition** *10-layer validation accuracy is bigger than 5-layer validation accuracy*
13: **save** 10-layer model results

The baseline algorithm is executed once and its results are stored for future comparison. For each type of convolutional layer, and for each type of activation function, the model is executed with one hidden layer. The model is then executed leveraging five hidden layers: if the accuracy results are better than the ones obtained when using a single hidden layer, these are stored; else, the model is executed again with a new condition obliging the accuracy to

be better than the one obtained with the one-layer model. The model is executing using ten hidden layers: if the accuracy results are better than the ones obtained when using five hidden layers, these are stored; else, the model is executed again with a new condition obliging the accuracy to be better than the one obtained with the five-layer model.

4.3 Analysis and Discussion of Results

The performance of the various GNN models, according to the convolutional layers applied and the activation functions, is detailed in the tables A.1 to A.32 in the appendix A[1]. The comparison of the results obtained for each model allows for the drawing of conclusions subordinated to four main scopes:

- Accuracy and Loss Values;
- Activation Functions;
- Number of Hidden Layers;
- Resource Consumption.

The comparison can also be checked in a more general, subjective way in table 4.4:

- Performance – evaluated according to the various values obtained for the statistic metrics (Accuracy, Precision, Recall);
- Convergence – assessed from the average times and epochs required to achieve results;
- Resource Consumption – estimated considering the percentage of CPU and RAM invested for the algorithm's execution.

In table 4.4, each convolutional layer is accompanied by the references to the four tables which present their results. The table's values are deduced from an high-level analysis of the obtained results.

Further analysis to the table shows some outliers or discrepant results which should be noted:

- Models with `AGNNConv` layers have slower convergence, especially when using the Sigmoid and Hyperbolic Tangent (TanH) activation functions (A.2,A.4);
- Models with `EdgeConv` layers tend to be unable to converge when applying ten hidden layers, except when applying the Sigmoid activation function (A.6);

[1]Let it be noticed that several tables have dashes in the values for the CPU and RAM percentages. This is due to no data being reported by the computer system itself, which prevents the values to be added.

Table 4.4: The general comparison between the models studied, grouped by convolutional layer type.

Model	Performance	Convergence	Resource Consumption
Baseline	High	Fast	Low
`AGNNConv` (A.1, A.2, A.3, A.4)	Good	Slow	Moderate
`EdgeConv` (A.5, A.6, A.7, A.8)	High	Slower, especially in deeper models	High
`GATConv` (A.9, A.10, A.11, A.12)	High	Fast	High, especially in terms of memory
`GeneralConv` (A.13, A.14, A.15, A.16)	Moderate, worst in deeper models	Stable, slow	Low
`GINConv` (A.17, A.18, A.19, A.20)	High	Stable, but poor in deeper models	High
`GraphSageConv` (A.21, A.22, A.23, A.24)	High	Moderate	Moderate
`GTVConv` (A.25, A.26, A.27, A.28)	High	Slower, with higher loss values	High
`TAGConv` (A.29, A.30, A.31, A.32)	High	Low training loss	Low

- Models leveraging `GATConv` layers only have quick convergence when using a single hidden layer of type TanH (A.12);
- Models using `GeneralConv` layers had better results with five hidden layers when leveraging the Sigmoid and TanH activation functions (A.14,A.16);
- Models composed of `GINConv` layers achieve higher accuracy when using less hidden layers, and only when applying ten hidden layers with the Soft-Plus activation function the accuracy value drops from 99.9% (A.19);
- Models leveraging `GraphSageConv` layers showed slower convergence when applying ten hidden layers with the Sigmoid and Soft-Plus activation functions (A.22, A.23);
- Models utilising `GTVConv` layers generally achieved high accuracy values with low elapsed times and number of epochs, except for the TanH activation function on ten hidden layers (A.28);
- Models composed of `TAGConv` layers generally achieved high accuracy results, with even lower elapsed times and number of epochs when compared to `GTVConv` models.

4.3.1 Accuracy and Loss

Most models achieved high training and testing accuracy values, which often exceeded the 99.9%.

The activation function that performed best was generally the Soft-Plus for every model. The best results of the Soft-Plus function occurred when leveraging `EdgeConv`, `GeneralConv`, and `GTVConv` layers, with values of 99.964, 99.929, and 99.953, respectively.

The Rectified Linear Unit (ReLU) and TanH activation functions showed slightly lower performance, with a decrease in accuracy as the number of hidden layers increased. `ReLU`'s best accuracy value is achieved when using `TAGConv` layers, 99.982. `TanH`'s best accuracy result is obtained with `GraphSageConv` layers, 99.905.

This can be noticed by looking at the values highlighted in bold on table 4.5.

Table 4.5: The general comparison in terms of accuracy, according to convolutional layer and activation function.

Convolutional Layer	ReLU	Sigmoid	Soft-Plus	TanH
Baseline	99.694			
`AGNNConv`	**99.893**	99.460	99.852	99.513
`EdgeConv`	99.929	99.941	**99.964**	99.941
`GATConv`	99.561	99.537	99.763	**99.780**
`GeneralConv`	99.911	99.905	**99.929**	99.893
`GINConv`	99.828	**99.947**	99.935	99.881
`GraphSageConv`	99.899	99.709	99.792	**99.905**
`GTVConv`	99.947	99.941	**99.953**	99.881
`TAGConv`	**99.982**	99.929	99.958	99.911

The activation functions which achieved the best results, ReLU, Soft-Plus, and TanH, were the ones which achieved the lowest loss values. This consistency of low loss values indicates a more stable training.

There are some exceptions to this conclusion, like the model using ten hidden `GATConv` layers with the Sigmoid activation function, which has a testing loss value of `6.882` (A.10), or the model using ten hidden `EdgeConv` layers with the ReLU activation function, which has a loss value of `10.133` (A.5). These values suggest potential issues like over-fitting, poor convergence, or numerical instability.

4.3.2 Activation Functions

The overall performance of the four activation functions considered for this project can be analysed as follows:

- ReLU – exhibited excellent performance with low loss and high accuracy values;
- Sigmoid – consistently achieved high accuracy and low loss, sometimes at the expense of longer training times;
- Soft-Plus – often led to higher loss and slightly lower accuracy values, especially when the models have more hidden layers;
- TanH – similarly to the soft-plus activation function, achieved worse results when dealing with models with a higher number of hidden layers.

As concluded in subsection 4.3.1 from table 4.5, the best activation function in general was Soft-Plus.

4.3.3 Number of Hidden Layers

In general, increasing the number of hidden layers did not translate into a consistent increase in the accuracy values. In some cases, like the model using `AGNNConv` layers with the Sigmoid activation function (A.2), or the model leveraging `TAGConv` layers with the TanH activation function (A.32), the accuracy decreases as the number of hidden layers increases. This implies there are some models in which over-fitting is likely to occur.

As expected, as the models use more hidden layers, the training time and computational resources invested increase. This is especially noticeable in models leveraging the `GATConv` and `GeneralConv` layers, which suggests that these might not scale efficiently.

The trade-off between accuracy gains and computational costs requires careful consideration when selecting the depth of the model, since more hidden layers do not necessarily mean higher accuracy values. In fact, as seen in table 4.6, most of the best accuracy results are obtained when using only one hidden layer.

Table 4.6: The comparison in terms of number of hidden layers of the models which achieved the best accuracy results, according to convolutional layer and activation function.

Convolutional Layer	**ReLU**	**Sigmoid**	**Soft-Plus**	**TanH**
`AGNNConv`	10	1	10	1
`EdgeConv`	5	1	1	1
`GATConv`	1	1	1	1
`GeneralConv`	10	5	10	5
`GINConv`	1	1	1	1
`GraphSageConv`	1	1	5	10
`GTVConv`	1	5	5	1
`TAGConv`	1	1	5	1
Most Common	1	1	*Ex-aequo* 1, 5	1

This predominance of the models with a single hidden layer can also be explained due to the construction of graphs prior to applying the GNNs. This facilitates understanding the connections between the services and verifying the calls within each trace execution, with no need to deepen or complicate the model itself to accommodate for these.

4.3.4 Resource Consumption

As expected, the computational resources invested in the execution of the baseline algorithm are less than the ones required for the running of the models applying hidden layers. Not only the existence of hidden layers implies more complexity within the model itself, but also the leveraging of different convolutional layers with various activation functions.

The usage of computational resources is fairly consistent throughout the various models tested in this project. The ReLU and Soft-Plus activation functions tend to have slightly higher CPU and RAM values compared to the Sigmoid and TanH, but this is a negligible difference. In fact, these values might be due to the remote connections made to the computer where the models were being executed. In general, the CPU values in training are closer to 87%, and the RAM values in training are close to 70%.

The training elapsed times have bigger importance than the testing elapsed times. The models which achieved better results with lesser number of epochs and lower training times leverage `EdgeConv` and `GINConv` layers with ReLU and Soft-Plus activation functions (A.5, A.19).

Models with lesser number of hidden layers tend to be executed faster. There are some exceptions like, for example, the model with one hidden layer of type `GeneralConv` with the Sigmoid activation function (A.14), or the model leveraging one hidden layer of type `EdgeConv` with the Sigmoid activation function (A.6). These can be considered outliers.

4.4 Addressing the Research Questions

This dissertation project aims to answer the two research questions proposed in the beginning, in section 1.2. The first question is answered in subsection 4.4.1, and the second question is answered in subsection 4.4.2.

4.4.1 First Research Question

The first research question is as follows: how to better the anomaly detection accuracy in the microservices system represented by the AIOps Challenge 2020 dataset using GNN?

As presented in section 2.2.3, a microservice system goes over a multitude of modifications during its lifecycle. Services are created and destroyed in a dynamic, fast, independent way, with multiple consecutive deploys. The malleability of this systems requires a constant updating of the monitoring and anomaly detection software to accompany and adapt to the changes it suffers.

By having a base which remains simple, a generic system can always be used with common parameters, like timestamp or unique identifiers of spans and traces[2]. This generalisation makes it possible to obtain results with high values – most of the accuracies obtained in this project are higher than 99.9%. Still, it is important to note that the probability of false positives is high, since there may be situations that are too specific to a system that are not picked up in a more generalised approach.

Detecting anomalies in a microservices system is a problem that can be tackled in many ways. GNNs are excellent at apprehending the interdependencies and interactions of a system's internal services. The use of graphs allows for faster and easier visualization of execution traces, allowing one to look at a set of processes as a whole.

4.4.2 Second Research Question

The second research question asks which number of hidden layers, and which type of convolutional layer, lead to the best classification results. To support this answer, the sections 4.3.1 and 4.3.3 provide the information required to answer it.

In terms of number of hidden layers, the best accuracies were achieved when using a single one, as seen in table 4.6.

In terms of activation functions, the last row of table 4.5 shows the best results were achieved when applying the Soft-Plus function.

In terms of convolutional layers, the last column of table 4.5 shows the best accuracy values were achieved when leveraging `EdgeConv` and `TAGConv` layers. The average accuracies for these types of layers were, respectively, 99.944% and 99.945%.

It is thus possible to conclude that the best model is composed of a single hidden layer of type `EdgeConv` or `TAGConv`, leveraging the Soft-Plus activation function.

4.5 Conclusion

Carefully considering the testing details, as discussed in section 4.2, guaranteed the required consistency and repeatability of the experiments executed. The reduction of variables, both environment and execution-wise, ensures the obtained results are as accurate as possible, allowing for their comparison among them, with the baseline algorithm as a benchmark.

The results obtained offer insights into the trade-offs between model complexity and resource usage. In general, the best model is the most simple one, achieving the best results with the least resource investment.

[2]Let it be noticed that, in this project, the service names are also used. This is not required in a generalised approach, but was implemented for the human visibility of the graph.

Chapter 5

Conclusion

5.1 Main Conclusions

Investigation, even if only at an academic level, is a full time job. Research involves a lot of behind the scenes work, mostly solitary and arduous. Research questions evolve as the investigation unfolds, and therefore it is common for the initial objectives of any project to change, partially or even totally, in its final version. This dissertation was no different, and the final objectives changed throughout the year.

These changes also imply differences in the final report, meaning some information is lost along the way. Nonetheless, there are some final conclusions that can be drawn.

Although various systems may have common characteristics, each of them is unique. This inherent specificity of each software prevents the existence of many approaches to anomaly detection that are applied globally, making it necessary to develop methods that are specific to each system to be monitored.

Likewise, each Machine Learning (ML) algorithm is unique and has more useful use cases for certain situations. It is necessary to carry out several experiments in order to have greater confidence in choosing the best algorithm to perform anomaly detection on each system.

The concepts studied, namely system architectures, microservices, and anomaly detection, allow for a more precise contextualisation of the problem. From these, the necessary foundations are obtained for the study of a new effective and efficient algorithm.

Another important factor to take into account is the discussion of microservice architecture and its performance. Microservices can be a very advantageous system architecture which allows companies to provide services with great flexibility and relatively low cost. Continuing to study microservices and anomaly detection algorithms will provide new data, and potentially find better solutions and approaches.

Generalised models achieve the best results when applying a lesser number of hidden layers, especially when leveraging Graph Neural Networks (GNNs). Having the connections between services and calls defined prior to the beginning of the execution allows for lesser complexity within the model itself.

Certain convolutional layers, like `EdgeConv` and `GTVConv`, have slower convergence. How-

ever, the worst results in terms of accuracy are achieved when applying convolutional layers of type `AGNNConv` and `GATConv`.

GNNs apply the power of ML in an easy, versatile way. There are multiple ways of implementing them, with more or less complexity, according to the necessities of the final solution. Utilising an already existent library like `Spektral` allows for faster implementation with good results and less effort.

5.2 Threats to Validity

Any project, and especially one of this scale, is bound to have issues which might threaten its validity, like:

- Algorithm's Specificity – an anomaly detection algorithm achieves its peak performance when built according to the system it will be monitoring. A more generalised approach, like this one, may achieve great results. However, these might overlook some niche occurrences which might be critical within the system;

- Testing Suite Size – although the AIOps Challenge 2020 dataset contains almost twenty million records of data, only thirteen occurrences represent anomalies. It does represent a real world system, however, this little number of anomalous spans might be only counted as false positive, instead of as a true positive, as intended;

- Baseline Comparison – most state-of-the-art algorithms and proposals are either unavailable or missing data, thus, the proposed algorithm cannot be put against their results directly.

5.3 Future Work

A project is never finished. This dissertation includes a lot of work, most of which cannot be reflected properly in this report.

This project can be enhanced with a bit more investment, investigating the results when:

- Applying other numbers of hidden layers, like two or three;

- Applying other types of layers, like pooling layers, which were not discussed in this document;

- Applying the algorithm to other datasets, like TrainTicket;

- Switching from a classification to a regression model, so the results are the probability of an occurrence being an anomaly.

Bibliography

[1] K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," *Journal of Management Information Systems*, vol. 24, pp. 45–77, 01 2007. xix, 4, 5

[2] M. Fowler and J. Lewis, "Microservices: A Definition of this New Architectural Term," https://martinfowler.com/articles/microservices.html, 2014, last accessed in 30 November 2023. xix, 1, 8, 9, 10, 11, 12

[3] M. H. Kashani, M. Madanipour, M. Nikravan, P. Asghari, and E. Mahdipour, "A Systematic Review of IoT in Healthcare: Applications, Techniques, and Trends," *Journal of Network and Computer Applications*, vol. 192, pp. 1–41, 10 2021. xix, 18

[4] O. Zimmermann, "Microservices Tenets: Agile Approach to Service Development and Deployment," *Computer Science - Research and Development*, vol. 32, pp. 301–310, 11 2016. 1, 10, 11

[5] F. Ponce Mella, G. Márquez, and H. Astudillo, "Migrating from Monolithic Architecture to Microservices," *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pp. 1–7, 09 2019. 1, 8, 10, 12

[6] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, and A. Barros, "Migrating Enterprise Legacy Source Code to Microservices: On Multi-Tenancy, Statefulness and Data-Consistency," *IEEE Software*, vol. 35, pp. 1–6, 06 2018. 1, 10

[7] S. Newman, *Building Microservices*, ser. ISBN: 9781492033998. O'Reilly Media, 2015. 1, 8, 9, 10, 11, 12, 19

[8] Coursera Staff, "What is a Service-Level Agreement? And How to Write One," https://www.coursera.org/articles/sla, 2023, last accessed in 04 April 2024. 1

[9] X. Zhou, X. Peng, T. Xie, J. Sun, W. Li, C. Ji, and D. Ding, "Delta Debugging Microservice Systems with Parallel Optimisation," *IEEE Transactions on Services Computing*, vol. 15, pp. 16–29, 05 2022. 1, 2, 14

[10] J. Bogatinovski, S. Nedelkoski, A. Acker, F. Schmidt, T. Wittkopp, S. Becker, J. Cardoso, and O. Kao, "Artificial Intelligence for IT Operations (AIOps) Workshop White Paper ," *ArXiv*, vol. abs/2101.06054, pp. 1–8, 01 2021. 1

[11] A. Dusia and A. S. Sethi, "Recent Advances in Fault Localization in Computer Networks," *IEEE Communications Surveys & Tutorials*, vol. 18, pp. 3030–3051, 5 2016. 1, 12, 13

[12] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs," *2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 683–694, 08 2019. 2, 14, 20, 22

[13] A. Shahid, G. White, J. Diuwe, A. Agapitos, and O. O'Brien, "SLMAD: Statistical Learning-Based Metric Anomaly Detection," *ICSOC 2020 Workshops – International Workshop on Artificial Intelligence for IT Operations*, vol. 12632, p. 252–263, 12 2020. 2, 19, 22

[14] H. Chen, K. Wei, A. Li, T. Wang, and W. Zhang, "Trace-based Intelligent Fault Diagnosis for Microservices with Deep Learning," *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 884–893, 07 2021. 2, 20, 22

[15] Z. Li, J. Chen, R. Jiao, N. Zhao, Z. Wang, S. Zhang, Y. Wu, L. Jiang, L. Yan, Z. Wang, Z. Chen, W. Zhang, X. Nie, K. Sui, and D. Pei, "Practical Root Cause Localisation for Microservice Systems via Trace Analysis," *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, pp. 1–10, 06 2021. 2, 19, 22

[16] D. Scheinert and A. Acker, "TELESTO: A Graph Neural Network Model for Anomaly Classification in Cloud Services," *Service-Oriented Computing - ICSOC 2020 Workshops*, vol. ISOC 2020, pp. 214–227, 05 2021. 2, 21, 22

[17] C. Zhang, X. Peng, C. Sha, K. Zhang, Z. Fu, X. Wu, Q. Lin, and D. Zhang, "DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-Based Deep Learning," *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, vol. ICSE '22, pp. 623–634, 05 2022. 2, 21, 22

[18] J. Chen, F. Liu, J. Jiang, G. Zhong, D. Xu, Z. Tan, and S. Shi, "TraceGra: A Trace-Based Anomaly Detection for Microservice Using Graph Deep Learning," *Computer Communications*, vol. 204, pp. 109–117, 04 2023. 2, 20, 22

[19] Z. Zeng, Y. Zhang, Y. Xu, M. Ma, B. Qiao, W. Zou, Q. Chen, M. Zhang, X. Zhang, H. Zhang, X. Gao, H. Fan, S. Rajmohan, Q. Lin, and D. Zhang, "TraceArk: Towards Actionable Performance Anomaly Alerting for Online Service Systems," *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 258–269, 05 2023. 2, 20, 22

[20] M. Li, D. Tang, Z. Wen, and Y. Cheng, "Microservice Anomaly Detection Based on Tracing Data Using Semi-supervised Learning," *2021 4th International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pp. 38–44, 05 2021. 2, 3, 19, 22

[21] C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu, "Eadro: An End-to-End Troubleshooting

Framework for Microservices on Multi-Source Data," *Proceedings of the 45th International Conference on Software Engineering*, vol. ICSE'23, pp. 1750–1762, 05 2023. 2, 20, 22

[22] A. for Computing Machinery, "ACM Computing Classification System," https://dl.acm.org/ccs, 2024, last accessed in 23 May 2024. 2

[23] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root Cause Localization of Performance Issues in Microservices," *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–9, 06 2020. 3, 13, 14

[24] X. Guo, X. Peng, H. Wang, W. Li, H. Jiang, D. Ding, T. Xie, and L. Su, "Graph-Based Trace Analysis for Microservice Architecture Understanding and Problem Diagnosis," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, vol. ESEC/FSE 2020, pp. 1387–1397, 11 2020. 3, 14, 16

[25] L. Meng, F. Ji, Y. Sun, and T. Wang, "Detecting Anomalies in Microservices with Execution Trace Comparison," *Future Generation Computer Systems*, vol. 116, pp. 291–301, 3 2021. 3, 14, 18, 19, 22

[26] D. Grattarola and C. Alippi, "Graph Neural Networks in TensorFlow and Keras with Spektral," *IEEE Computational Intelligence Magazine*, vol. 16, pp. 99–106, 02 2021. 3, 29

[27] H. Jaakkola and B. Thalheim, "Architecture-Driven Modelling Methodologies," *Frontiers in Artificial Intelligence and Applications*, vol. 225, pp. 97–116, 01 2010. 7

[28] International Business Machines Corporation, "What is SOA?" https://www.ibm.com/topics/soa, 2023, last accessed in 01 December 2023. 8, 12

[29] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance Engineering for Microservices: Research Challenges and Directions," *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pp. 223–226, 04 2017. 10, 11, 12

[30] C. Custer, "Vertical vs. Horizontal Scaling: What's the Difference and Which is Better?" https://www.cockroachlabs.com/blog/vertical-scaling-vs-horizontal-scaling/, 2023, last accessed in 15 April 2024. 11

[31] T. Yarygina and A. Bagge, "Overcoming Security Challenges in Microservice Architectures," *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pp. 11–20, 03 2018. 11

[32] P. Pushp, “Microservices Anomaly Detection,” https://www.linkedin.com/pulse/1-microservices-anomaly-detection-pushkar-pushp/, 2023, last accessed in 23 April 2024. 12

[33] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, “Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study,” *IEEE Transactions on Software Engineering*, vol. PP, pp. 243–260, 12 2018. 13, 14

[34] M. Jin, A. Lv, Y. Zhu, Z. Wen, Y. Zhong, Z. Zhao, J. Wu, H. Li, H. He, and F. Chen, “An Anomaly Detection Algorithm for Microservice Architecture Based on Robust Principal Component Analysis,” *IEEE Access*, vol. 8, pp. 226 397–226 408, 12 2020. 14

[35] P. Aggarwal, A. Gupta, P. Mohapatra, S. Nagar, A. Mandal, and Q. Wang, “Localization of Operational Faults in Cloud Applications by Mining Causal Dependencies in Logs Using Golden Signals,” *Service-Oriented Computing - ICSOC 2020 Workshops*, pp. 137–149, 05 2021. 14

[36] J. Hillpot, “4 Microservices Examples: Amazon, Netflix, Uber, and Etsy,” https://blog.dreamfactory.com/microservices-examples/, 2023, last accessed in 02 May 2024. 15

[37] M. Kolny, “Scaling Up the Prime Video Audio/Video Monitoring Service and Reducing Costs by 90%,” https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90, 2023, last accessed in 02 May 2024. 15

[38] T. Ng, “eBay Architecture,” https://www.slideshare.net/tcng3716/ebay-architecture, 2011, last accessed in 02 May 2024. 15

[39] D. Stenberg, “curl,” https://curl.se/, 2024, last accessed in 09 May 2024. 15

[40] M. Russinovich, “Microservices: An Application Revolution Powered by the Cloud,” https://azure.microsoft.com/en-us/blog/microservices-an-application-revolution-powered-by-the-cloud/, 2016, last accessed in 07 December 2023. 15

[41] Y. Izrailevsky, S. Vlaovic, and R. Meshenberg, “Completing the Netflix Cloud Migration,” https://about.netflix.com/en/news/completing-the-netflix-cloud-migration, 2016, last accessed in 05 December 2023. 15

[42] K. Goldsmith, “Microservices at Spotify,” https://www.youtube.com/watch?v=7LGPeBgNFuU, 2015, last accessed in 09 May 2024. 15

[43] Spotify, “Spotify Apollo Operations Manual,” https://spotify.github.io/apollo/, 2024, last accessed in 09 May 2024. 15

[44] A. Gluck, “Introducing Domain-Oriented Microservice Architecture,” https://www.uber.com/en-PT/blog/microservice-architecture/, 2020, last accessed in 05 December 2023. 16

[45] European Commission, “Technology Readiness Levels (TRL),” in *European Horizon 2020 - Work Programme 2014-2015*, 12 2015, p. 1, available online at https://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf. 16

[46] K. Cutajar and H. Xie, “How Prime Video Distils Time Series Anomalies into Actionable Alarms,” https://www.amazon.science/blog/how-prime-video-distills-time-series-anomalies-into-actionable-alarms, 2022, last accessed in 10 December 2023. 16

[47] Z. Zhang, K. Nie, and T. T. Yuan, “Moving Metric Detector and Alerting System at eBay,” *Association for the Advancement of Artificial Intelligence*, pp. 1–5, 04 2020. 16

[48] K. Lee, “Introducing 411: A New Open Source Framework for Handling Alerting,” https://www.etsy.com/codeascraft/introducing-411-a-new-open-source-framework-for-handling-alerting, 2016, last accessed in 05 May 2024. 16

[49] H. Ren, B. Xu, Y. Wang, C. Yi, C. Huang, X. Kou, T. Xing, M. Yang, J. Tong, and Q. Zhang, “Time-Series Anomaly Detection Service at Microsoft,” *The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, vol. 1906, p. 3009–3017, 08 2019. 17

[50] S. Esmaeilzadeh, N. Salajegheh, A. Ziai, and J. Boote, “Machine Learning for Fraud Detection in Streaming Services,” https://netflixtechblog.com/machine-learning-for-fraud-detection-in-streaming-services-b0b4ef3be3f6, 2022, last accessed in 02 May 2024. 17

[51] Lucas, “Can Spotify Detect Fake Streams?” https://themarketingheaven.com/can-spotify-detect-fake-streams/#Spotifys_Measures_Against_Fraudulent_Activity, 2024, last accessed in 04 May 2024. 17

[52] Y. H. Li, R. Agrawal, S. Shanmugam, and A. Pasqua, “Monitoring Data Quality at Scale with Statistical Modelling,” https://www.uber.com/en-PT/blog/monitoring-data-quality-at-scale/, 2020, last accessed in 10 December 2023. 17

[53] B. Kitchenham, “Procedures for Performing Systematic Reviews,” *Keele, UK, Keele University*, vol. 33, pp. 1–28, 08 2004. 18

[54] J. Lin, P. Chen, and Z. Zheng, “Microscope: Pinpoint Performance Issues with Causal

Graphs in Micro-service Environments," *16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings*, pp. 3–20, 11 2018. 19, 22

[55] Y. Jing, Q. Qi, J. Wang, T. Feng, and J. Liao, "ALSR: An Adaptive Label Screening and Relearning Approach for Anomaly Detection," *2019 IEEE Symposium on Computers and Communications (ISCC)*, vol. 136, pp. 1–6, 06 2019. 20, 22

[56] Y. Wang, Z. Wang, Z. Xie, N. Zhao, J. Chen, W. Zhang, K. Sui, and D. Pei, "Practical and White-Box Anomaly Detection through Unsupervised and Active Learning," *IEICE Transactions on Information and Systems*, pp. 1–9, 08 2020. 20, 22

[57] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, and D. Pei, "Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks," *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pp. 48–58, 10 2020. 21, 22

[58] Z. Xie, H. Xu, W. Chen, W. Li, H. Jiang, L. Su, H. Wang, and D. Pei, "Unsupervised Anomaly Detection on Microservice Traces using Graph VAE," *Proceedings of the ACM Web Conference 2023*, vol. WWW '23, pp. 2874–2884, 04 2023. 21, 22

[59] International Business Machines Corporation, "What is a data set?" https://www.ibm.com/docs/en/zos-basic-skills?topic=more-what-is-data-set, 2010, last accessed in 08 January 2024. 23

[60] Z. Li, N. Zhao, S. Zhang, Y. Sun, P. Chen, X. Wen, M. Ma, and D. Pei, "Constructing Large-Scale Real-World Benchmark Datasets for AIOps," *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 1–6, 08 2022. 23, 24

[61] S. G. Williamson and E. A. Bender, *Lists, Decisions and Graphs*, ser. ISBN: N/A. S. Gill Williamson, 2010. 23

[62] Z. Li and Z. Chen, "AIOps Challenge 2020 Data," https://github.com/NetManAIOps/AIOps-Challenge-2020-Data, 2022, last accessed in 08 January 2024. 24

[63] S. Jaiswal, "What is Normalisation in Machine Learning? A Comprehensive Guide to Data Rescaling," https://www.datacamp.com/tutorial/normalization-in-machine-learning, 2024, last accessed in 23 December 2024. 27

[64] S. Galli, "Exploring Oversampling Techniques for Imbalanced Datasets," https://www.blog.trainindata.com/oversampling-techniques-for-imbalanced-data/, 2023, last accessed in 23 December 2024. 27

[65] International Business Machines Corporation, "What is a Neural Network?" https://www.ibm.com/topics/neural-networks, 2024, last accessed in 25 April 2024. 27

[66] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, ser. ISBN: 9781292401171. Pearson Education, 2021. 27

[67] L. Wu, P. Cui, J. Pei, and L. Zhao, *Graph Neural Networks: Foundations, Frontiers, and Applications*, ser. ISBN: 9789811660535. Springer, 2022. 27, 28

[68] A. Amidi and S. Amidi, "Deep Learning Cheatsheet," https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning, 2018, last accessed in 28 April 2024. 28

[69] D. Grattarola, "Convolutional Layers," https://graphneural.network/layers/convolution/, 2024, last accessed in 25 April 2024. 29

[70] K. K. Thekumparampil, C. Wang, S. Oh, and L.-J. Li, "Attention-Based Graph Neural Network for Semi-Supervised Learning," *ArXiv*, vol. abs/1803.03735, pp. 1–15, 03 2018. 29

[71] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic Graph CNN for Learning on Point Clouds," *ACM Transactions on Graphics*, vol. 38, pp. 1–13, 01 2019. 29

[72] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," *6th International Conference on Learning Representations*, pp. 1–12, 02 2018. 29

[73] J. You, R. Ying, and J. Leskovec, "Design Space for Graph Neural Networks," *Proceedings of the 34th International Conference on Neural Information Processing Systems*, vol. NIPS '20, pp. 17 009–17 021, 12 2020. 29

[74] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How Powerful are Graph Neural Networks?" *7th International Conference on Learning Representations*, pp. 1–17, 07 2019. 29

[75] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," *31st Conference on Neural Information Processing Systems*, vol. NIPS '17, pp. 1025–1035, 12 2017. 29

[76] J. B. Hansen and F. M. Bianchi, "Total Variation Graph Neural Networks," *Proceedings of the 40th International Conference on Machine Learning*, pp. 1–24, 04 2023. 29

[77] J. Du, S. Zhang, G. Wu, J. M. F. Moura, and S. Kar, "Topology Adaptive Graph Convolutional Networks," *ArXiv*, 02 2018. 29

[78] D. M. Hawkins, “The Problem of Overfitting,” *Journal of Chemical Information and Computer Sciences*, vol. 44, pp. 1–12, 12 2003. 29

[79] A. Amidi and S. Amidi, “Machine Learning Tips and Tricks,” https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-machine-learning-tips-and-tricks, 2018, last accessed in 28 April 2024. 31

[80] W. Harris, “How the Scientific Method Works,” https://science.howstuffworks.com/innovation/scientific-experiments/scientific-method9.htm, 2024, last accessed in 10 October 2024. 34

Appendix A

Appendix

A.1 Results for Layer `AGNNConv`

Table A.1: Results obtained when using all hidden layers of type `AGNNConv`, using the Rectified Linear Unit (ReLU) activation function.

	Activation Function	**ReLU**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.768	99.838	99.812
	Loss (%)	0.585	0.439	0.512
	CPU (%)	86.918	85.557	85.164
	RAM (%)	78.774	79.814	78.818
	Elapsed Time (HH:MM:SS.mss)	01:00:13.733	03:09:31.541	02:51:52.306
	Epochs (#)	126	212	122
Testing	**Accuracy (%)**	99.774	99.875	99.893
	F-Score (%)	99.769	99.874	99.892
	Precision (%)	99.773	99.874	99.893
	Recall (%)	99.774	99.875	99.893
	Loss (%)	0.526	0.450	0.287
	CPU (%)	——	——	8.300
	RAM (%)	80.300	——	80.300
	Elapsed Time (HH:MM:SS.mss)	00:00:03.894	00:00:08.338	00:00:13.372

Table A.2: Results obtained when using all hidden layers of type `AGNNConv`, using the Sigmoid activation function.

	Activation Function	**Sigmoid**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.344	97.960	97.952
	Loss (%)	2.386	9.973	9.983
	CPU (%)	86.788	85.499	84.955
	RAM (%)	82.183	82.439	81.950
	Elapsed Time (HH:MM:SS.mss)	03:12:06.721	05:51:02.306	09:17:41.102
	Epochs (#)	400	400	400
Testing	**Accuracy (%)**	99.460	98.249	97.976
	F-Score (%)	99.420	97.381	96.974
	Precision (%)	99.456	96.528	95.992
	Recall (%)	99.460	98.249	97.976
	Loss (%)	1.729	8.843	9.899
	CPU (%)	——	——	——
	RAM (%)	87.200	86.700	87.100
	Elapsed Time (HH:MM:SS.mss)	00:00:03.993	00:00:08.330	00:00:13.500

Table A.3: Results obtained when using all hidden layers of type `AGNNConv`, using the Soft-Plus activation function.

	Activation Function	**Soft-Plus**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.772	99.830	99.841
	Loss (%)	0.579	0.490	0.433
	CPU (%)	86.730	85.517	85.198
	RAM (%)	74.009	77.185	81.934
	Elapsed Time (HH:MM:SS.mss)	00:57.32.342	04:48:46.288	08:48:06.288
	Epochs (#)	119	323	369
Testing	**Accuracy (%)**	99.715	99.840	99.852
	F-Score (%)	99.715	99.839	99.851
	Precision (%)	99.716	99.839	99.851
	Recall (%)	99.715	99.840	99.852
	Loss (%)	0.719	0.339	0.331
	CPU (%)	——	8.500	8.300
	RAM (%)	75.500	81.300	86.600
	Elapsed Time (HH:MM:SS.mss)	00:00:03.996	00:00:08.544	00:00:13.650

Table A.4: Results obtained when using all hidden layers of type `AGNNConv`, using the Hyperbolic Tangent (TanH) activation function.

	Activation Function	**TanH**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.360	99.334	99.351
	Loss (%)	3.003	3.959	3.911
	CPU (%)	86.836	85.508	84.969
	RAM (%)	82.102	82.148	75.044
	Elapsed Time (HH:MM:SS.mss)	03:11:28.661	05:49:31.294	09:18:24.108
	Epochs (#)	400	400	400
Testing	**Accuracy (%)**	99.513	99.442	99.430
	F-Score (%)	99.481	99.396	99.384
	Precision (%)	99.510	99.445	99.433
	Recall (%)	99.513	99.442	99.430
	Loss (%)	1.679	3.428	3.506
	CPU (%)	——	——	8.300
	RAM (%)	——	86.600	80.000
	Elapsed Time (HH:MM:SS.mss)	00:00:04.030	00:00:07.995	00:00:13.626

A.2 Results for Layer `EdgeConv`

Table A.5: Results obtained when using all hidden layers of type `EdgeConv`, using the ReLU activation function.

	Activation Function	**ReLU**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.870	99.850	98.016
	Loss (%)	0.461	0.287	11.875
	CPU (%)	87.088	86.124	85.818
	RAM (%)	70.663	76.807	76.323
	Elapsed Time (HH:MM:SS.mss)	00:31:12.186	01:14:34.840	17:06:44.360
	Epochs (#)	58	52	400
Testing	**Accuracy (%)**	99.887	99.929	97.916
	F-Score (%)	99.886	99.929	96.885
	Precision (%)	99.887	99.929	95.876
	Recall (%)	99.887	99.929	97.916
	Loss (%)	0.200	0.188	10.133
	CPU (%)	——	11.200	11.500
	RAM (%)	71.600	81.800	81.400
	Elapsed Time (HH:MM:SS.mss)	00:00:04.112	00:00:12.148	00:00:24.501

Table A.6: Results obtained when using all hidden layers of type `EdgeConv`, using the Sigmoid activation function.

	Activation Function	**Sigmoid**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.858	99.748	99.653
	Loss (%)	0.364	0.637	0.936
	CPU (%)	86.635	87.063	86.931
	RAM (%)	71.554	70.581	70.708
	Elapsed Time (HH:MM:SS.mss)	01:04:10.240	00:45:15.270	01:31:05.186
	Epochs (#)	117	31	35
Testing	**Accuracy (%)**	99.941	99.751	99.786
	F-Score (%)	99.940	99.743	99.781
	Precision (%)	99.941	99.750	99.786
	Recall (%)	99.941	99.751	99.786
	Loss (%)	0.210	0.495	0.574
	CPU (%)	——	11.500	12.000
	RAM (%)	——	71.000	71.200
	Elapsed Time (HH:MM:SS.mss)	00:00:04.082	00:00:12.048	00:00:22.973

Table A.7: Results obtained when using all hidden layers of type `EdgeConv`, using the Soft-Plus activation function.

	Activation Function	**Soft-Plus**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.905	99.736	96.670
	Loss (%)	0.228	10.829	21696215.625
	CPU (%)	85.128	88.806	88.801
	RAM (%)	71.371	76.544	76.895
	Elapsed Time (HH:MM:SS.mss)	01:10:18.600	10:59:57.354	20:04:01.241
	Epochs (#)	105	400	400
Testing	**Accuracy (%)**	99.964	99.739	98.017
	F-Score (%)	99.964	99.734	97.036
	Precision (%)	99.964	99.734	96.074
	Recall (%)	99.964	99.739	98.017
	Loss (%)	0.112	1.307	9.741
	CPU (%)	——	——	13.340
	RAM (%)	——	81.500	81.900
	Elapsed Time (HH:MM:SS.mss)	00:00:04.244	00:00:11.650	00:00:22.298

Table A.8: Results obtained when using all hidden layers of type `EdgeConv`, using the TanH activation function.

	Activation Function	**TanH**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.877	99.773	99.397
	Loss (%)	0.293	0.566	2.699
	CPU (%)	86.772	86.658	86.274
	RAM (%)	71.342	72.118	77.433
	Elapsed Time (HH:MM:SS.mss)	00:36:22.216	00:42:44.252	18:04:24.240
	Epochs (#)	65	29	400
Testing	**Accuracy (%)**	99.941	99.846	99.347
	F-Score (%)	99.941	99.844	99.285
	Precision (%)	99.941	99.846	99.351
	Recall (%)	99.941	99.846	99.347
	Loss (%)	0.195	0.364	2.413
	CPU (%)	——	11.100	11.850
	RAM (%)	72.200	72.600	83.200
	Elapsed Time (HH:MM:SS.mss)	00:00:04.086	00:00:13.056	00:00:24.692

A.3 Results for Layer `GATConv`

Table A.9: Results obtained when using all hidden layers of type `GATConv`, using the ReLU activation function.

	Activation Function	**ReLU**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.619	99.598	99.456
	Loss (%)	1.016	1.120	2.203
	CPU (%)	85.726	85.380	85.096
	RAM (%)	77.753	77.791	78.202
	Elapsed Time (HH:MM:SS.mss)	05:36:01.216	15:31:49.186	28:25:57.150
	Epochs (#)	400	400	400
Testing	**Accuracy (%)**	99.561	99.335	99.436
	F-Score (%)	99.535	99.277	99.395
	Precision (%)	99.561	99.340	99.426
	Recall (%)	99.561	99.335	99.436
	Loss (%)	5.929	6.192	22.576
	CPU (%)	——	9.100	9.250
	RAM (%)	82.900	82.800	83.400
	Elapsed Time (HH:MM:SS.mss)	00:00:06.844	00:00:22.340	00:00:41.605

Table A.10: Results obtained when using all hidden layers of type `GATConv`, using the Sigmoid activation function.

	Activation Function	**Sigmoid**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.608	99.518	98.540
	Loss (%)	1.056	1.541	6.882
	CPU (%)	85.801	85.670	85.202
	RAM (%)	74.138	78.041	75.067
	Elapsed Time (HH:MM:SS.mss)	05:37:48.222	16:05:36.301	28:54:34.324
	Epochs (#)	400	400	400
Testing	**Accuracy (%)**	99.537	99.216	98.072
	F-Score (%)	99.507	99.131	98.084
	Precision (%)	99.539	99.223	98.043
	Recall (%)	99.537	99.216	98.072
	Loss (%)	1.020	18.258	4.981
	CPU (%)	8.900	9.340	9.375
	RAM (%)	79.200	83.100	80.200
	Elapsed Time (HH:MM:SS.mss)	00:00:06.952	00:00:22.720	00:00:42.236

Table A.11: Results obtained when using all hidden layers of type `GATConv`, using the Soft-Plus activation function.

	Activation Function	**Soft-Plus**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.599	99.591	99.543
	Loss (%)	1.108	1.149	1.371
	CPU (%)	86.194	86.218	85.716
	RAM (%)	74.171	75.285	75.529
	Elapsed Time (HH:MM:SS.mss)	05:47:55.282	16:41:49.246	30:21:28.126
	Epochs (#)	400	400	400
Testing	**Accuracy (%)**	99.763	99.329	99.418
	F-Score (%)	99.763	99.263	99.367
	Precision (%)	99.763	99.334	99.422
	Recall (%)	99.763	99.329	99.418
	Loss (%)	1.020	4.554	4.589
	CPU (%)	——	9.700	9.775
	RAM (%)	79.200	80.300	80.700
	Elapsed Time (HH:MM:SS.mss)	00:00:06.985	00:00:23.064	00:00:42.933

Table A.12: Results obtained when using all hidden layers of type `GATConv`, using the TanH activation function.

	Activation Function	**TanH**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.582	99.460	99.204
	Loss (%)	1.135	1.596	3.707
	CPU (%)	85.837	85.550	85.174
	RAM (%)	73.412	74.851	75.080
	Elapsed Time (HH:MM:SS.mss)	00:35:53.210	15:51:05.306	28:48:25.288
	Epochs (#)	42	400	400
Testing	**Accuracy (%)**	99.780	99.448	97.803
	F-Score (%)	99.778	99.407	96.717
	Precision (%)	99.778	99.451	95.655
	Recall (%)	99.780	99.448	97.803
	Loss (%)	0.572	2.597	19.476
	CPU (%)	——	9.150	9.400
	RAM (%)	74.000	79.900	80.200
	Elapsed Time (HH:MM:SS.mss)	00:00:06.850	00:00:22.508	00:00:42.159

A.4 Results for Layer `GeneralConv`

Table A.13: Results obtained when using all hidden layers of type `GeneralConv`, using the ReLU activation function.

	Activation Function	**ReLU**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.797	99.843	99.790
	Loss (%)	0.487	0.378	0.537
	CPU (%)	87.206	87.278	89.071
	RAM (%)	71.508	71.252	71.116
	Elapsed Time (HH:MM:SS.mss)	00:50:21.300	00:52:58.312	00:46:12.276
	Epochs (#)	78	44	23
Testing	**Accuracy (%)**	99.780	99.899	99.911
	F-Score (%)	99.786	99.900	99.912
	Precision (%)	99.801	99.902	99.913
	Recall (%)	99.780	99.899	99.911
	Loss (%)	0.475	0.236	0.298
	CPU (%)	——	9.200	9.600
	RAM (%)	——	71.900	71.400
	Elapsed Time (HH:MM:SS.mss)	00:00:04.228	00:00:09.438	00:00:15.818

Table A.14: Results obtained when using all hidden layers of type `GeneralConv`, using the Sigmoid activation function.

	Activation Function	**Sigmoid**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.804	99.768	99.732
	Loss (%)	0.481	0.599	0.697
	CPU (%)	86.881	87.765	87.001
	RAM (%)	72.272	71.048	70.569
	Elapsed Time (HH:MM:SS.mss)	01:29:53.174	00:27:16.162	01:07:21.420
	Epochs (#)	135	21	31
Testing	**Accuracy (%)**	99.887	99.905	99.555
	F-Score (%)	99.888	99.906	99.578
	Precision (%)	99.890	99.909	99.640
	Recall (%)	99.887	99.905	99.555
	Loss (%)	0.334	0.312	0.897
	CPU (%)	——	9.600	10.000
	RAM (%)	74.100	71.400	71.000
	Elapsed Time (HH:MM:SS.mss)	00:00:04.362	00:00:09.821	00:00:16.659

Table A.15: Results obtained when using all hidden layers of type `GeneralConv`, using the Soft-Plus activation function.

	Activation Function	**Soft-Plus**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.695	99.821	99.820
	Loss (%)	0.828	0.429	0.438
	CPU (%)	87.542	89.034	89.525
	RAM (%)	71.086	71.396	71.755
	Elapsed Time (HH:MM:SS.mss)	00:23:01.138	01:08:27.481	02:23:27.138
	Epochs (#)	31	47	58
Testing	**Accuracy (%)**	99.852	99.858	99.929
	F-Score (%)	99.852	99.855	99.929
	Precision (%)	99.853	99.857	99.929
	Recall (%)	99.852	99.858	99.929
	Loss (%)	0.454	0.252	0.196
	CPU (%)	——	10.700	11.000
	RAM (%)	——	72.000	72.600
	Elapsed Time (HH:MM:SS.mss)	00:00:04.282	00:00:09.955	00:00:16.824

Table A.16: Results obtained when using all hidden layers of type `GeneralConv`, using the TanH activation function.

	Activation Function	**TanH**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.804	99.831	99.723
	Loss (%)	0.499	0.431	0.742
	CPU (%)	87.138	87.733	88.019
	RAM (%)	71.178	71.281	71.473
	Elapsed Time (HH:MM:SS.mss)	00:26:37.156	00:49:24.294	01:12:36.720
	Epochs (#)	41	39	34
Testing	**Accuracy (%)**	99.703	99.893	99.685
	F-Score (%)	99.692	99.894	99.673
	Precision (%)	99.704	99.895	99.686
	Recall (%)	99.703	99.893	99.685
	Loss (%)	0.489	0.275	0.828
	CPU (%)	——	——	9.800
	RAM (%)	——	71.800	72.000
	Elapsed Time (HH:MM:SS.mss)	00:00:04.260	00:00:09.641	00:00:16.363

A.5 Results for Layer `GINConv`

Table A.17: Results obtained when using all hidden layers of type `GINConv`, using the ReLU activation function.

	Activation Function	**ReLU**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.875	99.721	99.367
	Loss (%)	0.306	3.015	154.618
	CPU (%)	87.051	86.162	86.923
	RAM (%)	71.089	70.215	77.614
	Elapsed Time (HH:MM:SS.mss)	00:16:37.760	00:18:34.108	11:13:39.781
	Epochs (#)	37	18	15
Testing	**Accuracy (%)**	99.828	99.822	99.359
	F-Score (%)	99.824	99.824	99.300
	Precision (%)	99.828	99.830	99.363
	Recall (%)	99.828	99.822	99.359
	Loss (%)	0.344	0.421	3.818
	CPU (%)	——	——	9.700
	RAM (%)	——	70.500	83.400
	Elapsed Time (HH:MM:SS.mss)	00:00:03.736	00:00:08.290	00:00:14.243

Table A.18: Results obtained when using all hidden layers of type `GINConv`, using the Sigmoid activation function.

	Activation Function	**Sigmoid**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.846	99.782	99.808
	Loss (%)	0.391	0.588	0.489
	CPU (%)	85.843	85.252	88.586
	RAM (%)	71.761	71.118	71.811
	Elapsed Time (HH:MM:SS.mss)	00:38:29.228	00:21:56.126	01:38:32.228
	Epochs (#)	85	21	59
Testing	**Accuracy (%)**	99.947	99.846	99.774
	F-Score (%)	99.946	99.845	99.769
	Precision (%)	99.946	99.845	99.774
	Recall (%)	99.947	99.846	99.774
	Loss (%)	0.157	0.357	0.523
	CPU (%)	——	——	10.200
	RAM (%)	——	——	72.600
	Elapsed Time (HH:MM:SS.mss)	00:00:03.926	00:00:08.776	00:00:14.600

Table A.19: Results obtained when using all hidden layers of type `GINConv`, using the Soft-Plus activation function.

	Activation Function	**Soft-Plus**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.860	99.366	98.302
	Loss (%)	0.508	22.789	272.027
	CPU (%)	85.926	89.481	90.119
	RAM (%)	71.619	72.563	78.151
	Elapsed Time (HH:MM:SS.mss)	00:43:51.258	00:16:01.961	15:28:31.168
	Epochs (#)	74	12	400
Testing	**Accuracy (%)**	99.935	99.270	98.053
	F-Score (%)	99.935	99.198	97.089
	Precision (%)	99.934	99.275	96.143
	Recall (%)	99.935	99.270	98.053
	Loss (%)	0.111	4.281	9.599
	CPU (%)	——	——	11.100
	RAM (%)	——	72.800	83.700
	Elapsed Time (HH:MM:SS.mss)	00:00:03.929	00:00:08.917	00:00:15.041

Table A.20: Results obtained when using all hidden layers of type `GINConv`, using the TanH activation function.

	Activation Function	**TanH**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.847	99.778	99.672
	Loss (%)	0.383	0.557	0.866
	CPU (%)	85.887	85.688	86.702
	RAM (%)	71.103	71.223	73.012
	Elapsed Time (HH:MM:SS.mss)	00:18:15.108	00:36:55.216	00:37:50.222
	Epochs (#)	40	35	22
Testing	**Accuracy (%)**	99.881	99.733	99.780
	F-Score (%)	99.881	99.724	99.774
	Precision (%)	99.881	99.732	99.781
	Recall (%)	99.881	99.733	99.780
	Loss (%)	0.291	0.625	0.565
	CPU (%)	——	——	10.300
	RAM (%)	71.600	71.700	73.300
	Elapsed Time (HH:MM:SS.mss)	00:00:03.899	00:00:08.609	00:00:14.538

A.6 Results for Layer `GraphSageConv`

Table A.21: Results obtained when using all hidden layers of type `GraphSageConv`, using the ReLU activation function.

	Activation Function	**ReLU**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.814	99.839	99.817
	Loss (%)	0.486	0.466	0.543
	CPU (%)	79.213	75.148	74.267
	RAM (%)	75.164	73.363	73.606
	Elapsed Time (HH:MM:SS.mss)	01:57:59.342	01:32:58.192	03:08:53.480
	Epochs (#)	192	51	57
Testing	**Accuracy (%)**	99.715	99.852	99.899
	F-Score (%)	99.705	99.854	99.899
	Precision (%)	99.716	99.861	99.899
	Recall (%)	99.715	99.852	99.899
	Loss (%)	0.556	0.383	0.268
	CPU (%)	——	9.100	9.160
	RAM (%)	——	74.000	74.400
	Elapsed Time (HH:MM:SS.mss)	00:00:07.744	00:00:29.480	00:00:56.485

Table A.22: Results obtained when using all hidden layers of type `GraphSageConv`, using the Sigmoid activation function.

	Activation Function	**Sigmoid**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.672	99.315	97.965
	Loss (%)	0.895	2.359	9.961
	CPU (%)	78.886	75.661	75.254
	RAM (%)	75.331	73.359	78.042
	Elapsed Time (HH:MM:SS.mss)	02:09:45.541	01:13:32.781	23:17:16.102
	Epochs (#)	202	39	400
Testing	**Accuracy (%)**	99.709	99.567	97.976
	F-Score (%)	99.709	99.541	96.974
	Precision (%)	99.710	99.569	95.992
	Recall (%)	99.709	99.567	97.976
	Loss (%)	0.676	1.025	9.901
	CPU (%)	——	9.367	9.380
	RAM (%)	77.900	73.900	82.900
	Elapsed Time (HH:MM:SS.mss)	00:00:08.069	00:00:29.765	00:00:56.847

Table A.23: Results obtained when using all hidden layers of type `GraphSageConv`, using the Soft-Plus activation function.

	Activation Function	**Soft-Plus**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.740	99.538	97.956
	Loss (%)	0.679	1.146	10.001
	CPU (%)	81.317	78.262	78.186
	RAM (%)	75.295	73.442	78.122
	Elapsed Time (HH:MM:SS.mss)	02:05:31.300	01:30:46.180	26:34:46.204
	Epochs (#)	187	43	400
Testing	**Accuracy (%)**	99.745	99.792	98.076
	F-Score (%)	99.736	99.788	97.124
	Precision (%)	99.744	99.791	96.190
	Recall (%)	99.745	99.792	98.076
	Loss (%)	0.587	0.573	9.509
	CPU (%)	8.700	9.650	9.580
	RAM (%)	77.600	73.900	82.900
	Elapsed Time (HH:MM:SS.mss)	00:00:07.991	00:00:29.820	00:00:57.109

Table A.24: Results obtained when using all hidden layers of type `GraphSageConv`, using the TanH activation function.

	Activation Function	**TanH**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.851	99.866	99.847
	Loss (%)	0.394	0.346	0.397
	CPU (%)	79.503	75.368	75.303
	RAM (%)	74.344	73.683	73.842
	Elapsed Time (HH:MM:SS.mss)	01:10:34.600	02:11:46.661	03:06:24.360
	Epochs (#)	112	71	54
Testing	**Accuracy (%)**	99.840	99.893	99.905
	F-Score (%)	99.836	99.892	99.904
	Precision (%)	99.840	99.893	99.905
	Recall (%)	99.840	99.893	99.905
	Loss (%)	0.360	0.221	0.207
	CPU (%)	——	9.150	9.400
	RAM (%)	75.700	74.600	74.660
	Elapsed Time (HH:MM:SS.mss)	00:00:07.894	00:00:29.719	00:00:55.759

A.7 Results for Layer `GTVConv`

Table A.25: Results obtained when using all hidden layers of type `GTVConv`, using the ReLU activation function.

	Activation Function	**ReLU**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.917	99.820	99.778
	Loss (%)	0.198	0.460	0.573
	CPU (%)	86.953	87.932	87.470
	RAM (%)	74.216	73.298	73.489
	Elapsed Time (HH:MM:SS.mss)	01:17:03.102	01:07:13.421	02:05:29.300
	Epochs (#)	98	30	31
Testing	**Accuracy (%)**	99.947	99.798	99.893
	F-Score (%)	99.946	99.793	99.892
	Precision (%)	99.947	99.798	99.893
	Recall (%)	99.947	99.798	99.893
	Loss (%)	0.090	0.538	0.262
	CPU (%)	——	10.100	10.433
	RAM (%)	75.500	73.900	74.000
	Elapsed Time (HH:MM:SS.mss)	00:00:05.530	00:00:17.622	00:00:32.579

Table A.26: Results obtained when using all hidden layers of type `GTVConv`, using the Sigmoid activation function.

	Activation Function	**Sigmoid**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.842	99.815	98.960
	Loss (%)	0.398	0.473	4.079
	CPU (%)	86.413	87.306	84.661
	RAM (%)	75.223	74.461	72.947
	Elapsed Time (HH:MM:SS.mss)	02:21:04.126	04:40:33.240	25:24:58.144
	Epochs (#)	177	124	73
Testing	**Accuracy (%)**	99.935	99.941	99.858
	F-Score (%)	99.934	99.940	99.857
	Precision (%)	99.935	99.941	99.857
	Recall (%)	99.935	99.941	99.858
	Loss (%)	0.208	0.176	0.409
	CPU (%)	——	10.000	10.433
	RAM (%)	77.500	76.000	78.175
	Elapsed Time (HH:MM:SS.mss)	00:00:06.036	00:00:18.521	00:00:38.331

Table A.27: Results obtained when using all hidden layers of type `GTVConv`, using the Soft-Plus activation function.

	Activation Function	**Soft-Plus**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.892	99.818	99.796
	Loss (%)	0.291	0.467	0.504
	CPU (%)	85.192	88.739	88.408
	RAM (%)	73.496	73.256	73.778
	Elapsed Time (HH:MM:SS.mss)	00:32:13.192	00:47:46.282	04:11:42.661
	Epochs (#)	39	20	57
Testing	**Accuracy (%)**	99.881	99.953	99.923
	F-Score (%)	99.883	99.952	99.923
	Precision (%)	99.888	99.952	99.924
	Recall (%)	99.881	99.953	99.923
	Loss (%)	0.216	0.189	0.273
	CPU (%)	——	10.600	11.167
	RAM (%)	74.000	73.600	74.500
	Elapsed Time (HH:MM:SS.mss)	00:00:05.604	00:00:16.860	00:00:33.605

Table A.28: Results obtained when using all hidden layers of type `GTVConv`, using the TanH activation function.

	Activation Function	**TanH**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.811	99.757	99.073
	Loss (%)	0.459	0.591	5.005
	CPU (%)	86.655	87.733	85.310
	RAM (%)	74.012	73.451	72.985
	Elapsed Time (HH:MM:SS.mss)	01:00:38.894	01:40:21.240	25:01:33.606
	Epochs (#)	77	44	400
Testing	**Accuracy (%)**	99.881	99.774	99.472
	F-Score (%)	99.881	99.769	99.429
	Precision (%)	99.881	99.774	99.474
	Recall (%)	99.881	99.774	99.472
	Loss (%)	0.272	0.483	3.290
	CPU (%)	——	10.000	10.333
	RAM (%)	——	74.100	78.300
	Elapsed Time (HH:MM:SS.mss)	00.00:05.454	00:00:17.565	00:00:34.687

A.8 Results for Layer `TAGConv`

Table A.29: Results obtained when using all hidden layers of type `TAGConv`, using the ReLU activation function.

	Activation Function	**ReLU**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.905	99.880	99.830
	Loss (%)	0.224	0.293	0.469
	CPU (%)	83.619	83.068	80.785
	RAM (%)	73.999	73.604	73.521
	Elapsed Time (HH:MM:SS.mss)	00:39:04.234	01:38:51.228	01:05:12.301
	Epochs (#)	85	57	20
Testing	**Accuracy (%)**	99.982	99.881	99.893
	F-Score (%)	99.982	99.880	99.892
	Precision (%)	99.982	99.881	99.893
	Recall (%)	99.982	99.881	99.893
	Loss (%)	0.076	0.273	0.270
	CPU (%)	——	9.900	10.425
	RAM (%)	75.100	74.500	73.800
	Elapsed Time (HH:MM:SS.mss)	00:00:04.816	00:00:18.790	00:00:40.976

Table A.30: Results obtained when using all hidden layers of type `TAGConv`, using the Sigmoid activation function.

	Activation Function	**Sigmoid**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.864	99.841	99.734
	Loss (%)	0.342	0.426	0.845
	CPU (%)	82.735	82.685	80.828
	RAM (%)	74.219	73.459	73.425
	Elapsed Time (HH:MM:SS.mss)	00:46:15.276	01:17:47.102	01:43:41.258
	Epochs (#)	100	46	32
Testing	**Accuracy (%)**	99.929	99.911	99.810
	F-Score (%)	99.929	99.910	99.806
	Precision (%)	99.930	99.911	99.810
	Recall (%)	99.929	99.911	99.810
	Loss (%)	0.233	0.211	0.365
	CPU (%)	——	10.600	10.650
	RAM (%)	75.500	74.100	73.800
	Elapsed Time (HH:MM:SS.mss)	00:00:04.877	00:00:19.502	00:00:40.157

Table A.31: Results obtained when using all hidden layers of type `TAGConv`, using the Soft-Plus activation function.

	Activation Function	**Soft-Plus**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.908	99.855	99.842
	Loss (%)	0.221	0.350	0.410
	CPU (%)	84.163	85.752	85.258
	RAM (%)	73.849	73.340	73.702
	Elapsed Time (HH:MM:SS.mss)	00:40:27.240	00:55:51.330	01:51:58.306
	Epochs (#)	72	29	30
Testing	**Accuracy (%)**	99.929	99.958	99.893
	F-Score (%)	99.929	99.959	99.893
	Precision (%)	99.931	99.959	99.893
	Recall (%)	99.929	99.958	99.893
	Loss (%)	0.133	0.138	0.351
	CPU (%)	——	10.700	11.033
	RAM (%)	——	73.700	74.100
	Elapsed Time (HH:MM:SS.mss)	00:00:04.939	00:00:17.746	00:00:35.679

Table A.32: Results obtained when using all hidden layers of type `TAGConv`, using the TanH activation function.

	Activation Function	**TanH**		
	Hidden Layers (#)	**1**	**5**	**10**
Training	**Accuracy (%)**	99.843	99.811	99.634
	Loss (%)	0.381	0.454	0.947
	CPU (%)	83.562	82.731	80.950
	RAM (%)	73.675	73.254	73.589
	Elapsed Time (HH:MM:SS.mss)	00:27:53.162	00:42:26.252	01:08:47.480
	Epochs (#)	60	25	21
Testing	**Accuracy (%)**	99.911	99.863	99.739
	F-Score (%)	99.910	99.865	99.731
	Precision (%)	99.911	99.869	99.739
	Recall (%)	99.911	99.863	99.739
	Loss (%)	0.246	0.326	0.680
	CPU (%)	——	10.600	10.550
	RAM (%)	74.400	73.600	73.700
	Elapsed Time (HH:MM:SS.mss)	00:00:04.848	00:00:19.276	00:00:40.637