Graph Learning for Process Discovery

TESI DI LAUREA MAGISTRALE IN

COMPUTER SCIENCE AND ANGINEERING

|  |  |
| --- | --- |
| Student ID:  Advisor:  Co-advisor:  Academic Year: | 010499057  Matteo Matteucci  Pietro Portolani  2021/22 |

Author: **Matteo Badenchini**

# Abstract

**Keywords.**

# Abstract in lingua italiana

# Contents

[Abstract i](#_Toc115907218)

[Abstract in lingua italiana iii](#_Toc115907219)

[Contents 1](#_Toc115907220)

[Introduction 8](#_Toc115907221)

[1. Process Mining 10](#_Toc115907222)

[1.1 Overview and objectives 10](#_Toc115907223)

[1.2 Data 12](#_Toc115907224)

[1.3 Models 13](#_Toc115907225)

[1.3.1 Directly Follows Graphs 13](#_Toc115907226)

[1.3.2 Petri Nets and Workflow Nets 14](#_Toc115907227)

[1.3.3 Transition Systems 16](#_Toc115907228)

[1.3.4 Dependency Graphs and Causal Nets 17](#_Toc115907229)

[1.3.5 Process Trees 18](#_Toc115907230)

[2. Process Discovery 20](#_Toc115907231)

[2.1 Overview and objectives 20](#_Toc115907233)

[2.2 Challenges of Process Discovery 21](#_Toc115907234)

[2.3 Alpha relations and footprint 22](#_Toc115907235)

[2.4 Metrics 23](#_Toc115907236)

[2.4.1 Fitness 23](#_Toc115907237)

[2.4.2 Precision 25](#_Toc115907238)

[2.4.3 Simplicity 25](#_Toc115907239)

[2.4.4 Generalization 25](#_Toc115907240)

[2.5 State of the Art 26](#_Toc115907241)

[2.5.1 Algorithmic Miners 26](#_Toc115907242)

[2.4.2 Deep learning-based miner 31](#_Toc115907243)

[3. Deep Learning on Graphs 36](#_Toc115907244)

[3.1 Basic notions on Graphs 36](#_Toc115907246)

[3.2 Introduction to Deep Learning 38](#_Toc115907247)

[3.2.1 Structure 39](#_Toc115907248)

[3.2.2 Training 41](#_Toc115907249)

[3.2.3 Problems and mitigations 43](#_Toc115907250)

[3.3 Deep Learning on Graphs 45](#_Toc115907251)

[3.3.1 GAT 47](#_Toc115907252)

[3.3.2 GCN 48](#_Toc115907253)

[3.4 Deep Learning on Temporal Graphs 49](#_Toc115907254)

[3.4.1 TGN 50](#_Toc115907255)

[4. General Approach 53](#_Toc115907256)

[4.1 Difference with the State of the Art 53](#_Toc115907258)

[4.2 Most relevant failed attempts 54](#_Toc115907259)

[4.3 Final approach 56](#_Toc115907260)

[5. Data preparation 59](#_Toc115907261)

[5.1 Synthetic data generation 59](#_Toc115907263)

[5.2 Generation of the input model 60](#_Toc115907264)

[5.2.1 Transitions 60](#_Toc115907265)

[5.2.2 Places 61](#_Toc115907266)

[5.2.3 Bringing it all together 66](#_Toc115907267)

[5.3 Node features 66](#_Toc115907268)

[5.4 Train/test split 68](#_Toc115907269)

[6. Architecture, training, and inference 68](#_Toc115907270)

[6.1 Architecture 69](#_Toc115907272)

[6.2 Training 71](#_Toc115907273)

[6.3 Inference 71](#_Toc115907274)

[7. Results 72](#_Toc115907275)

[7.1 On synthetic data 72](#_Toc115907277)

[7.2 On real-life data 72](#_Toc115907278)

[8. Conclusion 72](#_Toc115907279)

[9. Bibliography 73](#_Toc115907280)

[10. List of Figures 75](#_Toc115907281)

# Introduction

In its broader definition, a process is a collection of activities that interact to produce a result. We find processes in many aspects of everyday life: when we order products online, when we apply for a job, when we assemble furniture, when we design and implement software. Special attention is given to processes that occur inside a company, the so-called business processes, in which a specific sequence of activities is executed to create a product or deliver a service to a customer. In this scenario, the company has every interest in designing efficient processes, or optimizing the ones already deployed. This is made possible by a thorough examination of the activities that take place inside the company: any delay, interruption or shortcut is recorded and used as a reference to optimize processes. Process Mining is the field devoted to such practice: it searches for patterns in the data generated by the execution of sequences of activities and gives practitioners the tools to study, optimize and check the processes under examinations.

The first step when studying a process is to reconstruct it from the recorded data, to have a clearer idea of what is happening inside the company and where the observed behavior diverges from the one expected. The practice of building the description of a process (also known as model) from the data coming from its execution is called Process Discovery, and Automated Process Discovery (or APD) when such discovery is made by means of algorithms or statistical techniques. In the last twenty years, the field has seen the rise of many different approaches, mainly based on algorithms (called miners), each one looking for specific patterns inside the data and yielding different model structures. Besides the progresses made, miners still suffer from problems related to the representation of the models they produce (e.g., whether they can detect non-local dependencies among activities), noise in the data and incompleteness of the datasets.

To limit such problems, recently the research has explored the benefits of introducing Machine Learning and Deep Learning into the miners. The result of such experiments is the miner realized by D. Sommers [1] in 2020: it works on graph-structured data, exploits Graph Neural Networks (GNNs) to learn patterns in the data and uses a stochastic approach to construct the model of a process. This method has the advantage of limiting the assumptions on the structure of the target model but relaxes some of the guarantees given by state-of-the-art algorithmic miners regarding the executability of the model.

In this document we will propose a different approach based on the guidelines defined by Sommers. Deep Learning and GNNs are still employed to learn patterns from data, but the resulting model is constructed following the temporal succession of the activities in the execution of the process.

This document is written according to the following structure:

* In chapter 1 we will give a more detailed description of Process Mining, discussing its goals, the tasks it can be divided into, the data it relies on, and the description of the processes (models) commonly used in the industry.
* In chapter 2 we will delve into the description of Process Discovery, by detailing common relations sought among the activities of a process, some of the most famous algorithmic miners available in commercial software and the most recent approach to APD using deep learning.
* In chapter 3 we will give a complete overview of Deep Learning, especially when applied to graph-structured datasets.
* In chapter 4 we will discuss the main differences between our work and the algorithm proposed by [1], as well as some key steps in the development of our algorithm. A more comprehensive description of the inner workings of our algorithm is given in chapters 5 and 6.
* In chapter 5 we describe the procedure to create a synthetic dataset and split it into train and test sets. We also outline the procedure to convert event logs into graphs that represent the input of our algorithm.

# Process Mining

In this chapter we will introduce the fundamental concepts related to the field of Process Mining. We will first discuss the role that Process Mining has in the industry, what are its goals and the main tasks it can be divided into. Then, we will focus on the type of data it deals with, where it is found and the information we can extract. Finally, we will give an overview of the main data structures used to model business processes, the so-called models.

## Overview and objectives

Process Mining can be defined as the bridge between data science and process science[[1]](#footnote-1) [2]. The goal of Process Mining is to discover, analyze and improve existing (business) processes by extracting the relevant information from event data stored inside information systems.

A business process is a collection of tasks and activities performed by people or (autonomous) systems that produces an outcome that contributes to the business goal. Processes are frequently described in form of graphs (e.g., Petri Nets), and this representation embodies the so-called model of the process. Event data is the result of the execution of such models.

Process Mining is one of the tools used to carrying out the Business Process Management (BPM) lifecycle, which describes the distinct phases of managing a particular business process.

Diagram

Description automatically generatedW. van der Aalst defines the BPM lifecycle as composed of three different phases [3]: *(re)design*,

Figure 1

In the *(re)design* phase, the process is designed in terms of its model, the abstraction that describes the interactions among its activities. Once designed, the model is implemented in the *implementation/configuration* phase by (re)organizing the working structure of the company and defining the roles and the (autonomous) agents needed for its execution. After that, the *run and adjust* phase begins: the model is deployed into production, executed, observed, and the event data it generates is collected. This data can then be used to extract the actual behavior of the process, which is later analyzed and compared against the expected behavior of the blueprint. Finally, if necessary, the process is redesigned by modifying its model, in the attempt to optimize (or remove) bottlenecks, include observed behavior and remove unobserved one.

The *run and adjust* phase is especially important, since it allows to close the cycle, and therefore to incorporate into the design information about the actual execution of the process in a systematic way.

In the field of Process Mining, we can identify three different branches (or type of tasks) that conduct the distinct phases of the BPM lifecycle: play-in, play-out and replay:

* Play-in, also known as discovery, refers to the practice of building the model of a process, starting from its observed behavior (event data). It represents the starting point of the BPM lifecycle, since it generates the model that is then put into production and subsequently analyzed and refined.
* Play-out consists instead in generating the behavior of a known process starting from the model that describes it. In this sense it reaches the opposite goal of play-in: it tries to produce event data (usually in form of logs) by simulating the model of the process.
* Replay is responsible for testing the model of a given process against the actual behavior observed. It serves different purposes: to spot bottlenecks, to find deviations from the blueprint and to try to predict problems that can arise from using such a model.

## Data

The starting point of any Process Mining task is event data, which can come, for example, from information and ERP systems, csv files, APIs, emails, or transaction logs. Data is usually gathered from different sources, and then refined and assembled into event logs, each one expressing the observed behavior of a different process.

Inside an event log, each entry represents a single event, which corresponds to the name of an activity (label) executed inside the process model. Besides the activity name, an event is usually characterized by other attributes, the timestamp and the case ID being among the most important ones. The case id denotes the unique identifier of a single run of the process model (case): a sequence of activities that, if performed, allows to go from the start of the process to the end of it. Each event describes the execution of a single activity and belongs to a single case, but different cases may contain the same activity (an example of this are the start and end activity, which are always executed for every case).

Other attributes (e.g., resource availability, cost, user/patience name) are usually present, and better characterize the execution of the process. Attributes can be at case level, if they are shared by all the events of the case (e.g., the patient’s name in case of medical records), or at event level, if they are unique for the event. Figure 2 shows an example of a real-life event log, in which besides case id, activity name and timestamp (*case item*, *event concept: name* and *event time: timestamp* respectively) we can see many more attributes that characterize the process.

A sequence of activities belonging to the same case is also known as a trace, and a log can be seen as a collection (or multiset) of traces. A trace, taken without its case, can appear multiple times inside a log, since different runs of a model can produce the same outcome reached by means of the same sequence of activities. A variant, instead, is a unique trace inside a log.

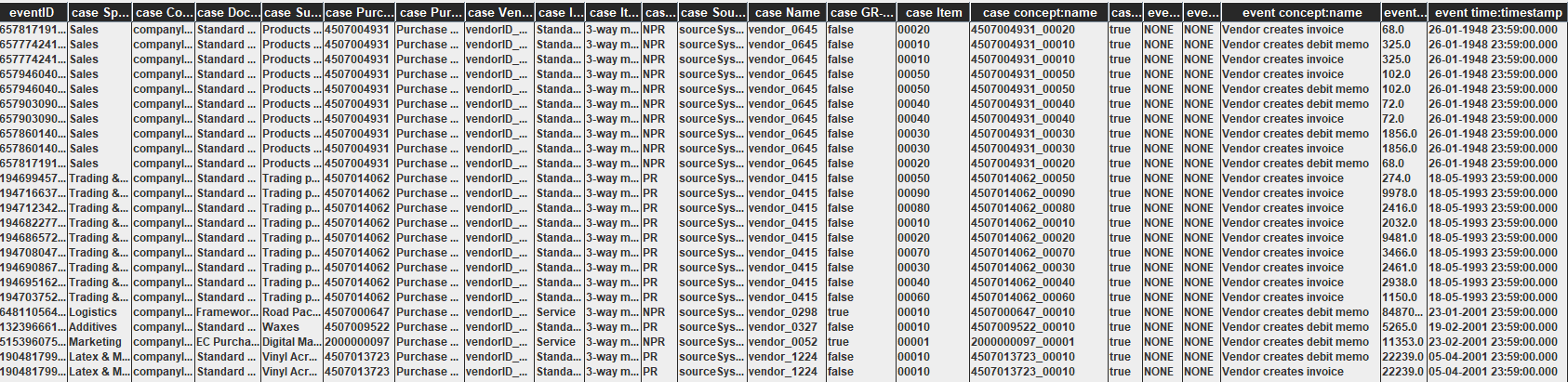
Let us clarify these definitions with an example. Suppose to have a log written in form of a multiset of traces (so a set in which each element is reported with its multiplicity), and be L = {<a, b, c, d>3, <a, c, b, d>2, <a, e, d>} such a log. Inside L we can identify three different variants, namely <a, b, c, d>, <a, c, b, d> and <a, e, d>, for a total of six traces.

Figure 2

## Models

Processes can be represented in a variety of forms, depending on one’s needs. In this paragraph we will discuss some of the most popular models, which can be either the final representation shown to the end user or the starting point (or the intermediate representation) of a discovery algorithm.

### Directly Follows Graphs

The directly follows graph (DFG for short) is one of the most straightforward representations of a process and the easiest to build starting from event logs. It consists of a directed graph whose set of nodes corresponds to the set of unique activity names in the event log. Arcs connect activities that directly follow each other inside a trace. The result is a model that captures the temporal relations among activities, but not necessarily causal dependencies (e.g., in case of parallel activities).

Logs can be filtered beforehand (by removing infrequent activities and/or traces) to build the model using only the most relevant information. This is done to prevent the construction of highly complicated models that include hundreds or thousands of arcs between activities. These complicated models are often referred to as *spaghetti models*, for their appearance resembles a plate of spaghetti (Figure 3).

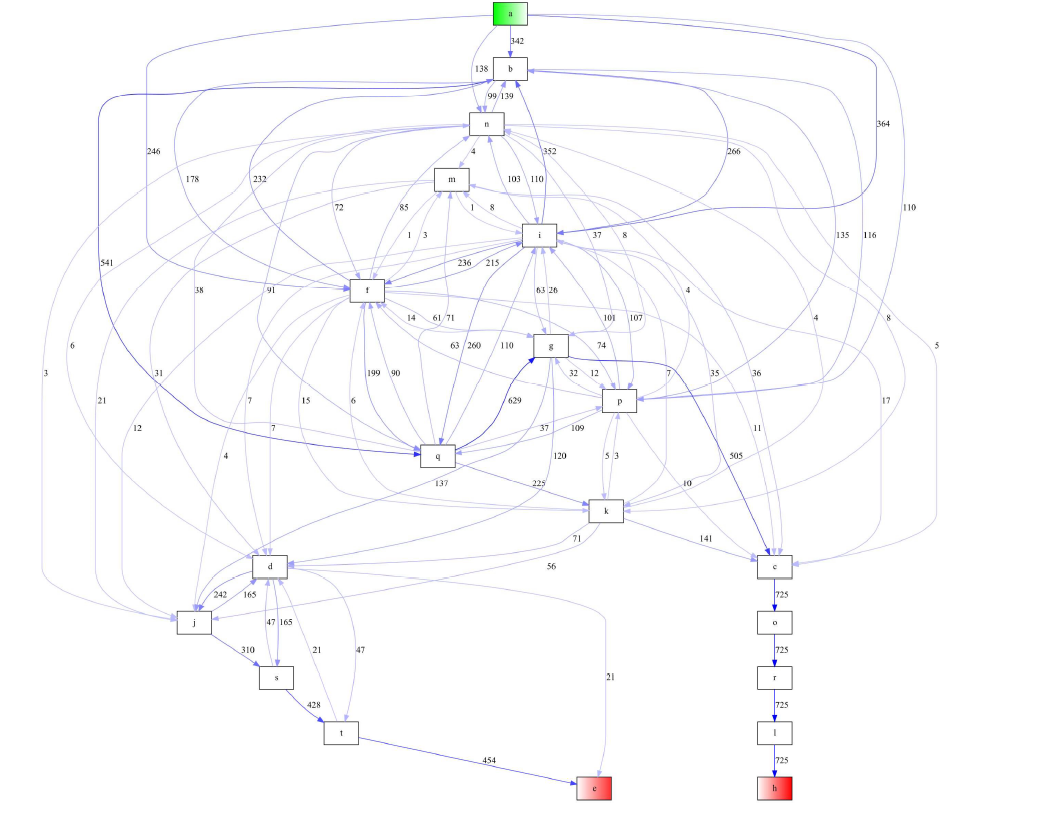


Figure 3

### Petri Nets and Workflow Nets

A Petri net is a bipartite and directed graph composed of places, transitions and arcs connecting places to transitions and vice versa. A place contains a given number of tokens, which can be considered as units of execution: as explained in more detail later, they are generated and consumed by transitions, and the changes in their distribution inside the net describe the different stages of the execution. Places are usually displayed as circles, transitions as boxes and tokens as black dots inside a place.

Transitions are usually associated with labels that describe the action being executed; when such a label is missing (or is denoted as τ), the transition is said to be *silent*, in the sense that its execution does not modify the behavior of the process described.

Diagram

Description automatically generatedIn the Process Mining context, transitions are associated with the activities of the log, while places are the nodes that control the order of execution of the activities by receiving tokens from (a subset of) the transitions in their backward star and passing them to (a subset of) the transitions in their forward star.

Figure 4

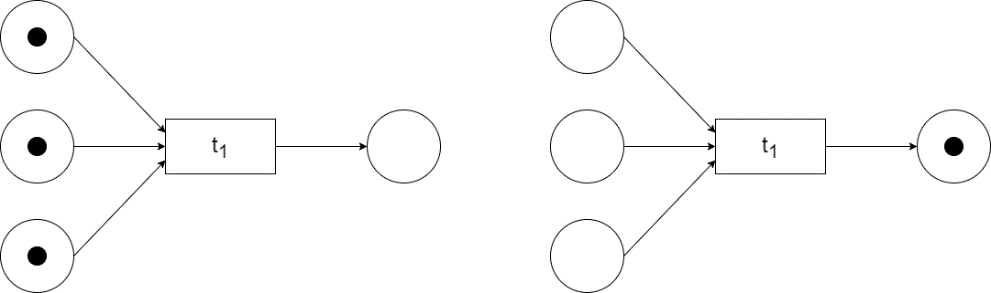


Figure 5

A transition is said to be enabled if each input place (those in its backward star) contains at least one token. When a transition is enabled, it can *fire* by consuming a token from each input place and producing a token for each output place. In Figure 4Figure 4, transition t1 is not enabled, since one of its input paces does not contain any token. In Figure 5, instead, we can see how the transition t1, once enabled, consumes all input tokens and produces a new token for each output place, in this example just one.

The distribution of the tokens inside a net is called marking and it defines the state of the net. A marking can be reachable or unreachable, depending on whether it is possible for the net to reach such a configuration starting from the initial state.

In principle, a place can contain an infinite number of tokens. We can however define an upper bound to such a number and say that a place is k-bounded if there is no reachable marking for which said place holds more than k tokens. A Petri net is defined k-bounded if all its places are k-bounded. When k is equal to one, the place (or the net, if the property holds for all its places) is said to be safe.

Petri nets can suffer from deadlock problems: these can arise whenever there is a reachable marking in which no transition is enabled[[2]](#footnote-2). The complete absence of such problems can be verified in live Petri nets, in which, for every transition, it is possible to reach a marking that enables it, starting from the initial reachable marking [4]. Note that the absence of deadlocks inside a Petri net is only a necessary condition for the liveness of the net; in this sense liveness implies the absence of deadlocks but the opposite is not true in general.

When a Petri net has a well-defined start (source) and end (sink) place, it is called workflow net (WF-net for short). Workflow nets are typically preferred when modeling a process, since the presence of a well-defined source and sink leads to models that are more easily interpretable by humans. Note that is always possible to transform a generic Petri net into a WF-net: it is sufficient to add two silent transitions, one firing all the starting places and one consuming the tokens coming from all the ending places. These two silent transitions are then connected, respectively, to a unique source and a unique sink.

A nice property of workflow nets is the soundness. A WF-net is said to be sound if and only if:

* It is safe: its places cannot hold multiple tokens at the same time.
* Allows for proper completion: if the sink is marked (meaning it contains a token), all other places are empty.
* It contains no dead parts: for any transition there is a firing sequence that enables it.

A workflow net is sound if and only if the corresponding short-circuited Petri net (a Petri net having a loopback (silent) transition) is live and bounded.

### Transition Systems

A transitions system is a type of model that tries to represent a process as a set of states connected to one another. As already discussed, a log can be seen as a set of traces, each of which specifies a possible execution of a model. In the context of a transitions system, the position of the execution inside a trace defines a state of the process, and the set of all possible states generated in this way (considering all the traces available for a given process) represents the set of nodes in the transition system.

Given two states, if it is possible to transition from the first state to the other by means of a single action, meaning by executing a single activity, the nodes corresponding to such states are connected. The arc connecting two states has a label that corresponds to the name of the activity that needs to be executed to transition from a state to the other.

A special kind of transition system is the *reachability graph* derived from a Petri net. In a reachability graph, each state corresponds to a marking of the corresponding Petri net, and the arcs connecting the states denote the activities that is necessary to execute to move from a marking to another; in this way, each firing sequence of the Petri net corresponds to a path in the corresponding reachability graph. This kind of transition system has a unique initial state (the marking that marks only the initial place(s)) but no explicit final state. Depending on the boundedness of the corresponding Petri net, a reachability graph can be finite of infinite: finite if the Petri net is k-bounded, infinite otherwise.

### Dependency Graphs and Causal Nets

A dependency graph is a directed graph that embeds the causal dependencies among activities in an event log. In this graph, each node corresponds to a unique activity in the event log, while the arcs connect nodes that are causally dependent from one another. Differently from directly follows graphs, two nodes are connected if and only if the corresponding activities are such that one enables the other, that is, given two activities A and B, A must be concluded before B starts, and the execution of B depends on whether A is executed in the first place.

Causal nets are strictly related to dependency graphs: nodes and arcs have the same meaning. However, in this model, each activity (node) has a set of possible input bindings and a set of possible output bindings. Given an activity A, an input binding represents a set of activities that enable A, while an output binding represents the set of activities that activity A can enable [5].

|  |  |
| --- | --- |
| Figure 6 | Figure 7 |

Figure 6 depicts an example of output bindings: activity *a* can enable either the set {*b, d*} or {*c, d*}, but not both. An output binding creates an obligation: once a binding is chosen, all the activities in it must be executed. Figure 7 describes an example of input bindings: activity *e* can be enabled by either the couple {*b, d*} or {*c, d*}, but not both.

Let us consider the bindings in Figure 6 and Figure 7, belonging to the came causal net: if activity *a* enables *b* and *d*, then activity *e* will be enabled by *b* and *d*, and not *c* and *d*. In other words, an obligation created by an output binding must be consumed by a (set of) input bindings for the activities involved.

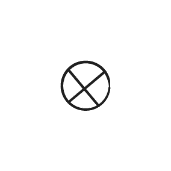
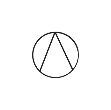
In a causal net the start activity has a no input bindings, while the end activity has no output bindings.

### Process Trees

Process trees are another powerful, yet easy to interpret, way of modeling a process. It describes a process as a tree whose leaves are the activities of process, and the internal nodes are operators connecting the different activities in a hierarchical way.

There are four possible operators, each of which defines a specific relation among its children:

* A picture containing shape

  Description automatically generatedSequence : n-ary operator; all the child nodes are are executed one after the other.
* Exclusive choice : n-ary operator; only one of the child nodes is executed.
* Parallel : n-ary operator; all the child nodes are executed.
* Loop A picture containing icon

  Description automatically generated : n-ary operator, with n ≥2; the leftmost child defines the condition, while the others constitute the body of the loop.

Process trees can be transformed into Petri nets by converting these operators into equivalent structures as we can see in Figure 8.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | | |
| Sequence PT to sequence Petri net structure | | | |
| A picture containing text, clock  Description automatically generated | | | Graphical user interface, application, Teams  Description automatically generated |
| Choice PT to choice Petri net structure | | | |
| Icon  Description automatically generated with medium confidence | | Application, icon  Description automatically generated | |
| Parallel PT to parallel Petri net structure | | | |
| Application  Description automatically generated | | Graphical user interface, application, Teams  Description automatically generated | |
| Loop PT to loop Petri net structure | | | |

Figure 8

# Process Discovery

In this chapter we will discuss the main ideas and objectives of process discovery, the branch of Process Mining devoted to the construction of models from event logs. Before delving into the description of some of the most famous algorithms, also known as miners, we will introduce the concept of alpha relations, which provides useful guidelines when trying to build Petri/Workflow nets, and the metric used to evaluate the goodness of a model. Then we will give an overview of some of the most famous algorithmic miners available in commercial tools, highlighting their strengths as well as their weaknesses. Lastly, we will describe the most recent attempt to the problem of process discovery: a deep learning-based miner that tries to overcome some of the limitations of its algorithmic counterparts by adopting a more human-like and explainable approach.



## Overview and objectives

As briefly discussed in the previous chapter, process discovery is the task of Process Mining that has the goal to map the content of an event log onto a process model (Petri net, process tree and so on) that is representative of the behavior seen in the event log.

Diagram

Description automatically generated

Figure 9

Ideally the result of such task is a sound workflow net that is able to replay all the traces seen in an event log without overfitting or underfitting. In other words, the resulting model should also have the ability to model unseen behavior, without generalizing too much. To further clarify this concept let us consider the log L = {<a, b, c, d>3, <a, c, b, d>2, <a, e, d>}. A model that can describe this log is for example the so-called *flower model* of Figure 9, that is considered to underfit the log since it can allow for the replay of any combination of traces that contain the activities {*a, b, c, d, e*}. On the other end, a model that enumerates all the possible traces in the log, such as the one depicted in Figure 10, is said to overfit the log, since it is only able to describe the content of the log from which it was built, without allowing for any other (unseen) behavior.

Shape

Description automatically generated

Figure 10

To assess the quality of the discovered model, four main metrics have been defined, namely fitness, precision, simplicity, and generalization, that we will describe in more detail further in this chapter.

## Challenges of Process Discovery

Process discovery is one of the most difficult tasks of Process Mining. Three are the main challenges that miners face when trying to discover a model from a log: incompleteness of the dataset, noise and what is called *representational bias*, so the set of the a priori assumptions made about the search space of the models.

For starter, logs only contain partial information about the process that generated them: capturing all the possible behavior would require infinite executions of the process, since only given enough time, a process will display every possible sequence of events. This is especially true for logs coming from real life data, since it is not infrequent for business processes to be abruptly interrupted by humans or faulty equipment, or for executed event not to be correctly recorded.

On a similar note, the problem of the noise is related to the fact that usually logs contain infrequent behavior that is not representative of the main characteristics of a process.

Lastly, as briefly mentioned, there is the problem of representational bias. Miners are designed to extract relevant information from logs by looking for specific patterns and converting them into meaningful structures in the target model; that is to say, they have to make assumptions about the search space of the possible models. An example in which representational bias can occur, is for example when a miner only tries to discover a model by allowing for activities to only appear once in the structure, while another would be making decisions about the way activities are connected based only on local interaction and not looking at the process as a whole.

All miners suffer from such problems. The difference is in the way they deal with them and produce results that are acceptable enough for the industry.

## Alpha relations and footprint

To define the structure of a Petri/Workflow net from the information contained inside event logs, we first need a way to establish an ordering of its transitions. Alpha relations are a set of rules designed to achieve this goal; they work by classifying the relationship between couples of activities into four categories based on their relative position inside an event log. These relations are:

* Direct succession, denoted with x > y. Given two activities *x* and *y*, they are in direct succession if and only if there exist a case inside the log for which *x* is directly followed by *y*.
* Causality, denoted with x 🡪 y. Given two activities *x* and *y*, they are causally related if and only if x > y and not y > x. In other words, if x precedes y for some cases, the inverse should not be true for any other case.
* Parallel, denoted with x//y. Given two activities *x* and *y*, they are parallel if and only if x > y and y > x. This relation describes concurrency inside the log, the situation in which two activities are executed simultaneously; because of that, there is no fixed ordering for termination of the activities, and they can appear in either configuration of the directly follow relation.
* Choice, denoted with x # y. Given two activities *x* and *y*, they are in a choice relation if and only if not x > y and not y > x. In this situation, the two activities cannot be both activated from the same source, but a choice must be taken.

These relations are used by some discovery algorithms as a reference to build the final model, while others only exploit them for preprocessing the output, as we will discuss in this chapter.

The set of all the relations of a log is the *footprint* of the log. If we consider for example the log L = {<a, b, c, d>3, <a, c, b, d>2, <a, e, d>}, we can see that , , , , , and , while all the other possible relations are of type choice. Based on that, the footprint of this log is the following:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e |
| a |  |  |  |  |  |
| b |  |  |  |  |  |
| c |  |  |  |  |  |
| d |  |  |  |  |  |
| e |  |  |  |  |  |

Where the symbol indicates the specular relation with respect to .

## Metrics

As mentioned above, there are four main criteria that allow practitioners to assess the quality of the discovered models: fitness, precision, simplicity, and generalization. In loose terms, fitness measures how much the behavior observed in a log can be replayed by the model, in terms of percentage of traces correctly executed; precision on the other hand, can be seen as a limitation on the behavior allowed by the model - a precise model would generalize to similar behavior, while a flower model for example could produce any combination of the considered activities; simplicity is an indication of the complexity of the result, usually in terms of the number of connections among activities; generalization indicates how well a model can produce unseen (plausible) behavior[[3]](#footnote-3).

### Fitness

Fitness is usually described by means of one of the following metrics: token-based replay fitness [6] and alignment-based replay fitness [7]. The former is model dependent and tries to execute each trace of the log on the model, forcing the execution by adding tokens when necessary and keeping track of such modifications. The latter is model independent and guarantees that semantically equivalent models will have the same fitness values.

Let us say we want to evaluate a Petri net. To compute the token-based fitness, we need to keep track of how the markings of the net change during the execution, and in particular the number of tokens produced *p*, the number of tokens consumed *c*, the number of missing tokens *m* and the number of remaining tokens *r* after the execution. Every time a transition fires, *p* is incremented by a value equal to the number of places reached by the transition; every time a transition is enabled, *c* is incremented by a value equal to the number of input places for that transition; every time a token needed to carry out the execution is missing, such token is introduced in the net and *m* incremented by one; at the end of the replay, *r* is set equal to the number of tokens remaining inside the net. Then, the metric is computed according to the following equation:

where L denotes the log, N the model being evaluated, denotes a trace in the log and the frequency of such trace.

In the alignment-based approach we try to find, for each trace in the log, the corresponding sequence of activities through the model, going from beginning to end, such that this sequence is as close as possible to the trace (or optimal). For a Petri net, such a sequence of activities matches to a path in the reachability graph of the net, such that the labels of the arcs traversed correspond to the transitions that have fired in the net during the execution. To choose the optimal sequence, a ranking of all the possible sequences constructed on the activities of the trace is computed; this ranking is based on a cost function that can be as simple as the edit distance[[4]](#footnote-4).

For the alignment-based approach to work, *relaxed soundness* [8] is required: for each activity in the model, it should exist at least one firing sequence that connect the starting activity of the process to the end activity of the process.

Given a trace, the optimal sequence is found, as well as the worst possible one, that is the one that differs from the trace in every position. The fitness for that trace is computed as

where L denotes the log, N the model being evaluated, denotes a trace in the log, the frequency of such trace and and are, respectively, the costs of the optimal and worst sequence for each trace.

### Precision

Similarly to what happens for the fitness, precision can be computed using two main approaches, one token-based [9] and one alignment-based [10], and their inner workings are like those described above. The idea behind the two approaches, however, is similar: for each trace, the different prefixes are replayed on the model, whenever possible. At each reached marking (or position in the alignment), the set of transitions that are enabled in the process model is compared to the set of activities that follow the prefix: the more the sets are alike, the higher is the precision.

### Simplicity

Simplicity tries to describe the complexity of the solution in terms of the arc degree of the nodes in the model [11]. If we consider a Petri net, we compute the average degree for each place and transition of the net, defined as the sum of the output and input arcs of each node divided by the number of nodes. Then the simplicity of the model is given by

where N is the model considered and k a parameter decided by the user such that .

### Generalization

Generalization measures how well the model can describe the unseen behavior of a process, and not only the one displayed in the event log [12]. It is based on how often the different parts of the model are used during the replay of the log: if some parts are infrequently visited, generalization is bad, while if all parts are frequently used generalization is likely to be high. If we imagine the model being a process tree, then the formula to compute the generalization if the following:

## State of the Art

In this section we will introduce some of the most famous miner used in commercial tools, the algorithmic miners, and the state of the art for what concerns the process discovery method based on deep learning.

### Algorithmic Miners

#### Alpha algorithm

The Alpha algorithm was, historically, the first miner to be able to discover the concurrency of activities inside a process. It works by taking in input an event log, and after constructing the footprint, it directly exploits the alpha relations among activities to build a Petri net; consequently, the produced model generates the same footprint of the log.

The first step is to extracts all the activity names from the log and use them to construct the set T of all the transitions of the model. Once T is available, it proceeds to define the sets TI and TO of starting and ending activities.

The second and most important step is to construct the places of the model. At this stage, the places are defined as pairs of sets of transitions (A, B) such that, for every transition ta in A there exist a transition tb in B that directly follows ta, and all the transitions in A and B respectively are in choice relation among each other. In symbols:

Where XL is the set of all such pairs. This set is then filtered by keeping only the pair of sets that are maximal, so the pairs such that by removing any element of A or B, the condition stated before no longer holds; these remaining pairs are the final places of the model. For each place, A contains the incoming transitions, while B the outgoing transitions, so the model is built accordingly.

|  |  |
| --- | --- |
| Diagram  Description automatically generated  Figure 11 | A picture containing text, sport  Description automatically generated  Figure 12 |

In its basic form, this algorithm has many limitations:

* It is unable to detect loops of length ≤ 2. If not taken care of in pre or post processing, this leads to unconnected transitions that, according to the semantics of Petri nets, are always enabled.
* It can generate implicit places: places that connect transitions that if removed do not alter the behavior of the net. This problem can be easily solved in post processing.
* By rigidly following the alpha relations, it is unable to detect nonlocal dependencies. Alpha relations are based on the directly follows relation, so if in the real process there are two activities are such that one eventually follows the other (meaning they are at distance ≥ 1), this dependency is not identified by the Alpha algorithm.
* It cannot identify free-choice constructs, such as those in Figure 11 and Figure 12.
* It suffers from representational bias: the assumption of unique activities (so no transitions with the same label are allowed) and the absence of OR-split and joins (that cannot be detected by the alpha relations). Both these problems could be solved with the introduction of silent transitions.
* The resulting model is not guaranteed to be sound.
* It is not robust to noise (infrequent traces or activities), so preprocessing may be needed.

Besides its limitations, the Alpha algorithm represents a good baseline when doing process discovery.

#### Heuristics Miner

The Heuristic Miner [13] works in two phases: at first it learns a dependency graph from the input log; then it transforms the graph into a causal net by learning the appropriate bindings by means of some predefined heuristics.

The dependency graph is built according to two indicators: the direct succession , and the dependency measure , defined as follows for two generic activities *a* and *b* in the log L:

where denotes a trace in the log, the frequency of such trace and the i-th activity of trace . As we can see, the dependency measure is based on the direct succession, and it can be observed that , with a value of meaning strong causality among the two activities. These two measures are used to prune the original dependency graph constructed directly from the log, by keeping only the arcs for which and are above a certain predefined threshold.

Once the dependency graph is available, the bindings are learnt by means of some heuristics (e.g., use a time window around each activity to keep only the most frequent activities among the one closest in time in the log) or optimization criteria (e.g., define a function to rank the possible combination of bindings and keep the best ones) that depend on the version of the miner.

The resulting causal net can be converted to a Petri net, but in this case the Petri net is not guaranteed to be sound. Beside this problem, since the two indicators used by the algorithm in the first phase are based on the direct succession relation, the Heuristic miner suffers from some of the limitations that afflict the Alpha miner, for example the inability to detect nonlocal dependencies among activities. When used on large datasets, this miner also frequently leads to spaghetti models, which are difficult to interpret.

#### Region-Based Miner

The Region-Based miner, as the Heuristic one, works in two steps: it first learns an intermediate representation of the model and then based on this intermediate representation it builds the final model. In this case the intermediate representation is a transition system, while the final model a Petri net.

The most crucial decision in the construction of the transition system regards the way the states are defined: they can embed information about the frequency and the order of the activities in the log, for example by means of a time window around each activity (that can be symmetric or focused just on the past or future interactions of the activity) or, in a more abstract way, consider the states as multisets of activities, each one reported with the frequency with which it appears inside a particular trace or the whole log. This decision has an important impact on the final model: if the states include information about the activities’ order, for example, the resulting model could overfit, while more abstract representations help mitigate the problem.

Once the transitions system is defined, the algorithm proceeds to identify regions inside it. A region, in this context, is defined as a set of states of the transition system, such that if a transition exists the region, then all equally labeled transitions exit the region, and if a transition enters the region, all equally labeled transitions enter the region[[5]](#footnote-5). In particular, the algorithm is interested in finding all the non-trivial minimal regions inside the transition system. A trivial region is a region that contains all the states of the transition system or that contains none, while a minimal region is a region that cannot be decomposed into smaller, non-trivial ones.

The set of all the minimal, non-trivial regions constitutes the set of all the places in the Petri net; the set of all the transition labels in the transition system instead defines the set of all the transitions of the Petri net. Once places and transitions are defined the arcs connecting them are generated.

As already mentioned, this approach can produce overfitting models, depending on the definition of state for the transition system, but is able to discover complex process patterns that are undetected by the miners previously described. Other limitations regard the number of minimal non-trivial regions that can be found: if none is found, the result is an unconnected Petri net, in which all transitions are always enabled. In general, a transition in the resulting Petri net can be disconnected from the rest of the net.

#### Inductive Miner

The inductive miner [14] is one of the best performing algorithmic miners available, being able to guarantee the soundness of the output model. It works by iteratively partitioning the directly follows graph into subsets of activities and using these subsets to build a process tree.

The criteria followed to partition the directly follows graph are closely related to the definition of the four possible operators in a process tree:

* The *sequence cut* partitions the directly follows graph into subsets of nodes such that every activity in one subset follows every activity of another subset based on the direct succession relation. The subsets generated are linked by the *sequence* operator.
* The *exclusive choice* cut generates partitions that are completely disconnected (the activity in each partition do not interact with the activities of any other partition). The subsets generated are linked by the *choice* operator.
* The *parallel cut* generates partitions such that the activities contained in each subset can follow each other in any order (e.g., if two subsets A and B are found, it is possible for any activity inside A to follow and be followed by any activity inside B). The subsets generated are linked by the *parallel* operator.
* The *redo-loop cut* finds a partition in which its subsets are such that the execution of the activities inside some of the subset is conditioned on the execution of the activities of other subsets. For example, let us consider three subsets A, B and C that are found by this cut. If the execution of any activity in B or C is conditioned on the execution of A, then A represent the condition of the loop, while B and C the body of the loop. The subsets generated are linked by the *loop* operator.

|  |  |
| --- | --- |
| Diagram  Description automatically generated  Figure 13 | Diagram, engineering drawing  Description automatically generated  Figure 14 |

In Figure 13 it is possible to see an example directly follows graph, and Figure 14 the resulting process tree found by the miner[[6]](#footnote-6). In this example is easy to see how the sets {*a}* {*b, c, f, e*} and {*d*} can be individuated using the *sequence cut*. The same cut can be applied to the subset {*e, f*}, while the subset {*b, c*} can be partitioned using the *exclusive choice* cut. Finally, subsets {*b, c*} and {*e, f*} can be individuated using the *redo-loop* cut.

The resulting process tree can be translated into a sound Petri net, by introducing silent transitions when needed.

Limitations of this miner are mainly due to its representation bias: the leaves must have a unique label (so there cannot be repetitions of activities in different areas of the resulting model) and the process must be described with a block-structured representation. Also, fitness and precision can be increased by increasing the number of silent transitions inside the final model, that has the side effect of reducing the simplicity of the model.

### Deep learning-based miner

So far, state-of-the-art techniques have approached the problem of process discovery as an unsupervised learning problem, through graph synthesis algorithms such as those discussed above. This approach has the downside of introducing biases in the resulting models, since each algorithm tries to find a solution by means of different design criteria and/or heuristics [15]. Recently, to overcome this issue, researchers have tried to switch to a more human-like approach by leveraging the ability of Neural Networks to learn patterns through data alone. The most famous work in this area is the one by D. Sommers [1], who formulated the problem as a supervised learning problem: a deep learning-based algorithm that tries to learn how to build process models from examples of (synthetic) event logs and corresponding template models (search space) and then transfers this knowledge to build models for real-life unseen logs. This algorithm takes in input a graph that encodes the traces of a given event log and the template of a Petri net (more on that later), it elaborates this input by means of subsequent passes through graph neural networks and shallow multilayer perceptrons, and outputs a Petri net which is a subgraph of the one encoded in the input.

#### Data

The first step is to encode the process discovery problem as a graph. This graph is a directed multigraph composed of two parts: a bidirectional graph that encodes the traces of the event log, called the trace graph, and the template of a Petri net, built according to the alpha relations discovered in the log and augmented with a subset of all the possible places that connect the transitions.

Shape, box and whisker chart

Description automatically generated

Figure 15

The trace graph, of which we can see an example in Figure 15, consists of different chains of nodes, one for each variant in the event log, each node representing an activity of the process (except the initial and final nodes, that represent the fictitious start and end activities). These chains are linked to two artificial nodes marking the start and the end of the traces. The features of each node are initialized with a one-dimensional tensor holding the one-hot encoding for that node in the first n-1 places, and the frequency of the trace in the nth place.

The Petri net template, instead, is built starting from the alpha relations discovered in the event log. The transitions of this net consist in the set of the process activities, to which are added the transitions corresponding to an artificial start and end activities. Places are introduced according to the following criteria:

1. The places associated with the directly follows relations are added. These places are referred to as one-to-one places.
2. Places with a single incoming transition and multiple outgoing transitions are added. These correspond to the places which connect activities that are directly followed by more than one activity. Let us consider for example an event log in which activity *a* is directly followed by activities *b*, *c*, and *d*, and there is no parallel relation among *b*, *c* and *d*. In this case, the places that will be added to the Petri net are 4 and correspond to all the possible combinations of the three activities (so the 4 possible output configurations for activity *a*): {*b*, *c*}, {*b*, *d*}, {*c*, *d*}, {*b*, *c*, *d*}. [figure x] shows an example of this. In the eventuality in which two or more of the following activities are parallel to each other, the places that express the combinations in which such activities are present are discarded (so, continuing the previous example, if *c*//*d*, the only places we add are {*b*, *c*} and {*b*, *d*}). A similar methodology is applied to add all places that have more than one incoming transition and only one outgoing transition. These kinds of places are respectively referred to as one-to-many places and many-to-one places.
3. In the last step, many-to-many places are introduced, so places that have multiple common incoming and outgoing transitions. The criteria followed to create them is like the one in step three but combines the two ideas.

The features associated to the places are initialized to zero. Transitions and places are then connected with directed arcs that go in both directions, allowing for the information to flow in the graph without any limitations (e.g., a preferred direction).

Once the two subgraphs are constructed, they are joined by means of directed arcs that connect the nodes of the trace graph to the corresponding transitions of the template Petri net with the same name.

#### Approach

Given the graph obtained during preprocessing, the goal is to learn to select the subset of places that restricts the transitions of the template Petri net to the behavioral patterns present in the log (sequence, parallel and so on). This is achieved by estimating the likelihood that a specific place , where P is the set of all the places in the template Petri net, is best suited to be included into the result given that places have already been selected by the miner, i.e., . The actual likelihood, however, is dependent on the ordering π in which the nodes are considered, and so the joint probability that the places in are selected is defined as with ordering π. Since the order is not relevant to the solution, we can use the marginal joint probability to model the probability over all permutations of : . can then be used to estimate the best subset of places in the template net.

Since it is not actually feasible to compute the marginal joint probability over all the possible permutations of , the actual problem of the computation is addressed in two different ways at training and inference time. During training, teacher forcing is used to impose a breadth first order on the miner, that gives a lower bound on the actual marginal joint probability. In this way, the model learns the joint distribution , with π being the breadth first ordering of the places in the template Petri net. The miner is then trained by maximizing the expected joint log likelihood , with and from dataset , where and are respectively a log and the corresponding input graph constructed as discussed above. With a sufficiently large and varying dataset, approaches , which is the true probability distribution of the correct selection of places from *P* given an event log L.

During inference, since teacher forcing is not available, beam search is applied to keep track of multiple candidate sequences of places and mitigate the effect that suboptimal choices in the beginning can have on the final sequence. Then the most promising one(s) is (are) considered as the result(s) of the discovery.

#### Structure of the miner

The functioning of this miner is possible thanks to its structure. It is composed of four different neural networks: two graph neural networks called *propagation networks* and two shallow multilayer perceptrons, the *select candidate network* and the *stop network*. The details on how a graph neural network and a multilayer perceptron are structured will be given in the following chapter: for now, it is sufficient to consider them as black boxes that, given an input, are able to learn patterns and produce an output.

The first propagation network takes in input one of the graphs constructed during the preprocessing phase and returns an initial embedding of its nodes. After that, the core of the miner is executed: a do-while cycle controlled by the *stop network* evaluates, one at a time, a candidate place among those in *P*, and the decision on whether to include it in the final result is made by the *select candidate network*. Before the condition of the cycle is evaluated, the embeddings of the nodes are further preprocessed by the second *propagation network* that produces new embeddings that contain the information about the decisions made so far by the *select candidate network*. When the *stops network* ends the cycle, the template Petri net is extracted from the processed graph and returned to the user.

#### Results

When compared with other algorithmic miners[[7]](#footnote-7), this miner has demonstrated to be able to reach similar results to the state of the art. It is not guaranteed to generate sound models, not even relaxed sound, but its result yield in general to high precision, generalization, and simplicity (where it outperforms the state of the art), but in general a lower fitness. When tested on synthetic data, it generates models that are closer to the ground truth than those produced by the other miners, thus outperforming them; on real data, the results are comparable to the state of the art.

# Deep Learning on Graphs

In this chapter, we will address how to use Deep Learning techniques to operate on graph-structured data. We will first introduce some notions about graphs, their properties and the tasks that can be tackled on such data structures; then we will present the basic concepts about neural networks and Deep Learning, with a brief discussion on the training procedure, the concept of overfitting and some best practices on how to avoid it. After that, we will move on to examine the basic building blocks that can be used to build a neural network that operates on graphs (Graph Convolutional Networks and Graph Attention Networks) and the challenges of Deep Learning on this kind of data structure. Lastly, the topic of Deep Learning on temporal graphs is addressed, with a focus on Temporal Graph Networks.



## Basic notions on Graphs

In its simplest form, a graph can be described as a collection of nodes (or vertices) and edges (or arcs). Nodes are usually visualized as circles with some data inside of them, while edges, that represent the connection among nodes, are thought of as lines that connect a pair of nodes. If the direction of the edges in a graph matters, the graph is said to be directed (and the edges are represented as arrows), otherwise it is undirected (respectively Figure 16 and Figure 17). Graphs give a useful framework to represent entities and the relations among them: for example, nodes can be cities and edges the roads that connect them; or we can model the relationships between people if we think of them as nodes and the interactions among them as edges.

|  |  |
| --- | --- |
| A picture containing text, pool ball, vector graphics  Description automatically generated  Figure 16 | A picture containing text, pool ball, vector graphics  Description automatically generated  Figure 17 |

More formally, a graph G is a tuple , where V is the set of nodes and E the set of edges, such that with . A graph in which there is at most one edge between each pair of nodes, no edges between a node and itself and all edges are undirected is called *simple*.

One of the most common ways to represent a graph is by means of an *adjacency matrix* : if two nodes and are connected, then cell contains a non-zero value[[8]](#footnote-8). The adjacency matrix is symmetric when the graph is undirected: since the order of the edges is not important, it must be that and , with .

In general, a pair of nodes can be linked by more than one edge, and when this happens, we talk about multi-relational graphs: in such graphs, edges are described as a triple , where τ is the edge (or relation) type. The presence of multiple types of relations, makes it necessary the definition of multiple adjacency matrices , one for each possible type.

Multi-relational graphs are mainly found in two configurations: heterogeneous or multiplex graphs (or a combination of the two). In heterogeneous graphs, nodes also have types, and so we can partition the set of nodes into disjoints sets . Edges then satisfy the constraints according to the node type: . A special case of heterogeneous graphs are multipartite graphs, where edges only connect nodes of different types: . Petri nets are an example of multipartite graphs with only two sets of nodes (bipartite). In multiplex graphs, instead, graphs can be decomposed in a set of *k* layers: each node is assumed to belong to every layer, and each layer represent a unique relation. Intra-layer edges connect nodes at the same layer, while inter-layer edges are used to connect the same node across different layers. An example of multiplex graph is the map of the subway: intra-layer edges connect stations that are reachable from the same line, while inter-layer edges are represented by the different subway lines available at each station (e.g., if station A can be reached by line 1 and line 2, these two lines represent the inter-layer connection that allows a person to reach A with line 1 and leave with line 2).

Besides the information about the type of node or edge, we can associate attribute or feature information to a graph, and this can be done at node, edge, or graph level: features at node-level, for example, are described as , where |V| is the number of nodes in the graph and *m* the number of features for each node.

Typical Machine/Deep Learning tasks on graphs are: node classification, when we want to predict the label associated to a node given the labels of a subset of nodes in the graph; relation/link prediction, when given a set of nodes and an incomplete set of edges we want to infer the presence (or absence) of the missing edges; clustering and community detection, when we want to identify subsets of nodes in a graph that share some similarities. Similar tasks are also defined at graph level, when for example we want to predict some properties of a graph or cluster graphs according to their similarity.

## Introduction to Deep Learning

Machine learning (ML) is the subfield of Artificial Intelligence devoted to the study and development of methods that perform specific tasks without being explicitly programmed to do so. A Machine Learning algorithm learns how to perform such tasks from the data by optimizing an object function (or performance measure). A more formal definition, given by Tom Mitchell, is the following: *a computer is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.* The spam filter in an email client is an example of ML algorithm that tries to distinguish “good” emails from “spam”: the *experience* is built on the collection of emails available at a given point in time, the *task* is to classify a new email as “good” or “spam”, and the *performance measure* the accuracy with which a new email is classified.

Such tasks can be learned in a *supervised* or *unsupervised* manner. A *supervised algorithm* learns by confronting the prediction it makes against the *ground truth* (the correct answer), and adjusting the future predictions based on the errors (and correct answers) made during its *training*. An *unsupervised algorithm* learns the task to perform without knowing the correct answer, but by finding associations among datapoints based on a function that is problem dependent.

Deep Learning (DL) is a subset of ML methods based on Artificial Neural Networks (ANN or simply NN). The goal is to use such networks to approximate an unknown differentiable function *f*, and this is done by learning from the data how to correctly produce outputs that are as close as possible to the outputs of the function. DL is commonly used when dealing with problems for which there is not a clear (mathematical) formulation, like classifying the content of an image and translating human languages. Similarly to other ML tasks, neural networks can be trained in a supervised or unsupervised manner: an example of the former is the image classification task, where usually the ground truth is available and the neural network can learn to adjust its output based on the correct answer; an example of the latter, instead, is the representation learning task, where the goal of the network is to learn a lower-dimensional representation of the input so that it contains all the important features necessary to reconstruct it.

### Structure

The building block of an ANN is the *neuron*, a function that approximate the biological cell it is named after. Like a real neuron, the neuron of an ANN receives some inputs, elaborates them, and produces an output signal that is received by another neuron.

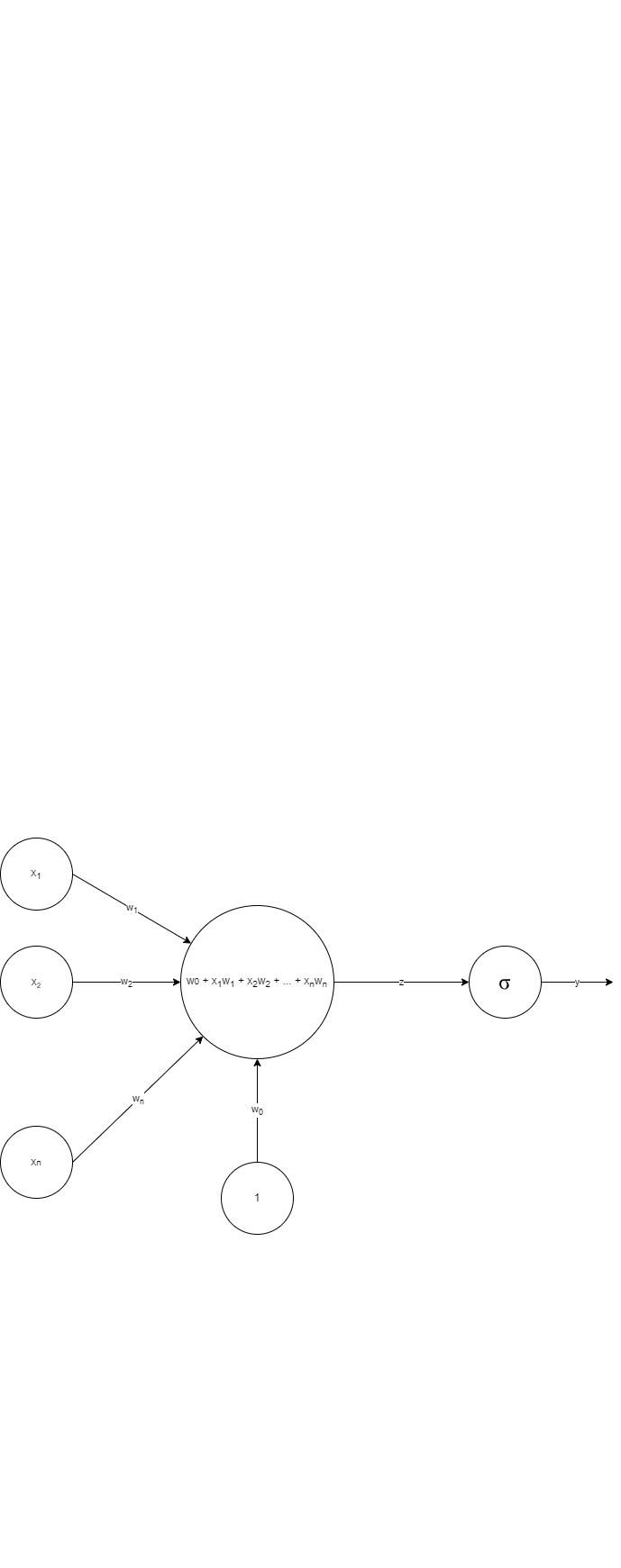


Figure 18

As we can see in Figure 18, that depicts a graphical representation of a neuron, each input (also called *feature*) is associated with a weight , a learnable parameter that the neural network adjusts during training; the neuron uses such weights to compute a weighted sum of the inputs and produce an intermediate signal . is then fed into a nonlinear function (also called *activation function*), that has the purpose of extending the expressive power of the neural network, which otherwise will produce outputs that are a simple linear combination of the inputs.

Mathematically, a neuron is represented by the following equation:

where is the activation function, the intermediate signal and a special weight that is not associated to any input, called *bias*.

Neurons are stacked together to form complex architectures, like the one in Figure 19, and those that share the same inputs and outputs are said to belong to the same *layer*. A neural network can be seen as a stack of different layers, each one with a given number of neurons inside; all the layers in between the input and output ones are called *hidden*.

The structure of a neural network is usually referenced to as *architecture*, while a particular instance of this architecture (so with trained weights) is called *model[[9]](#footnote-9)*. The simplest (and historically the first) architecture of a neural network consists of layers of densely connected neurons[[10]](#footnote-10), and is called Multilayer Perceptron, MLP for short (Figure 19). More complex architectures have been developed to tackle different problems. One of the most widely used is the encoder-decoder architecture, which is composed of two parts: the encoder that elaborates the initial features of the input sample and condenses them into a low-dimensional representation called *embedding*; the decoder, that perform problem-specific tasks starting from the embedding previously produced. Example of this architecture can be seen in models used in image classification tasks, where the encoder produces a condensed representation of an image, and the decoder classifies the content on the image from the information contained in the embedding.

Background pattern

Description automatically generated

Figure 19

### Training

Neural networks learn the most suitable parameters for their neurons during training, a process during which the network is exposed to a subset of all the data available, called *training set*, and iteratively updates the weights according to a measure of the difference of the network’s output and the expected one, called *error*. This error is computed according to a special function, called *loss*, that is problem dependent.

A pass over the entire dataset is called *epoch*, and typically the training of a NN consists of several epochs.

The training process of a neural network is seen as an optimization problem: the goal is to change the weights of the network so that the error associated to the network’s outputs is as small as possible. This is done by means of an algorithm called *backpropagation* that, for each sample, computes the gradient of the loss function with respect to the weights of the network. The computation is done efficiently by leveraging on the chain rule of the derivatives[[11]](#footnote-11), and this efficiency allows for the use of gradient methods, such as the *gradient descent*, to update the weights and minimize the error. When using gradient descent, the update rule of the weights is as follows:

where is the loss function, the weight that is being updated and an hyperparameter called *learning rate*. The learning rate has the goal of adjusting the step of the update: if is too big the error could diverge in future epochs, since the weights will reach big values; if it is too small, the error will not diverge, but the training will take longer because the steps will be closer to zero for small values of the error. A small learning rate can also cause the algorithm to reach a local optimum, with the steps being too small to escape such point.

Other gradient methods commonly used for this purpose are the *stochastic gradient descent* and the *batch gradient descent*. In the former, the network’s weights are updated after each training sample; this leads to a number of iterations per epoch equal to the number of samples and tends to be noisy (but is optimal when working with a small memory). In the latter, the update is done by averaging the error of a *batch* of points; this gives more stability to the evolution of the error and requires less iterations of the algorithm per epoch (but consumes more memory, since it needs to keep the values of the partial updates before averaging them).

Beside gradient descent, other algorithms can be used during the training of a neural network, depending on one’s needs (e.g., the need for an adaptive learning rate).

### Problems and mitigations

For supervised learning algorithms in ML and DL, the generalization error is a measure of how well an algorithm can generalize its predictions on unseen data. A typical way to analyze such error is the bias-variance decomposition, that sees the generalization error as a sum of three terms: bias, variance, and irreducible error. Bias measures the deviations between expected and actual outputs and describes how well the algorithm is able to fit the training data. Variance, on the other hand, describes how much the predictions of the algorithm change when applied on unseen data. The third term, the irreducible error, is an indication of the noise in the prediction that results from the problem itself and cannot be eliminated. The only two terms on which we can operate to reduce the error are, therefore, bias and variance. As we can see in Figure 20, minimizing both is not trivial: in general, when bias increases variance decreases, and vice versa. This is known as the *bias-variance tradeoff*. The goal, when training an algorithm, is to find the spot in the curve for which bias and variance are low enough to make the algorithm perform well on unseen data.

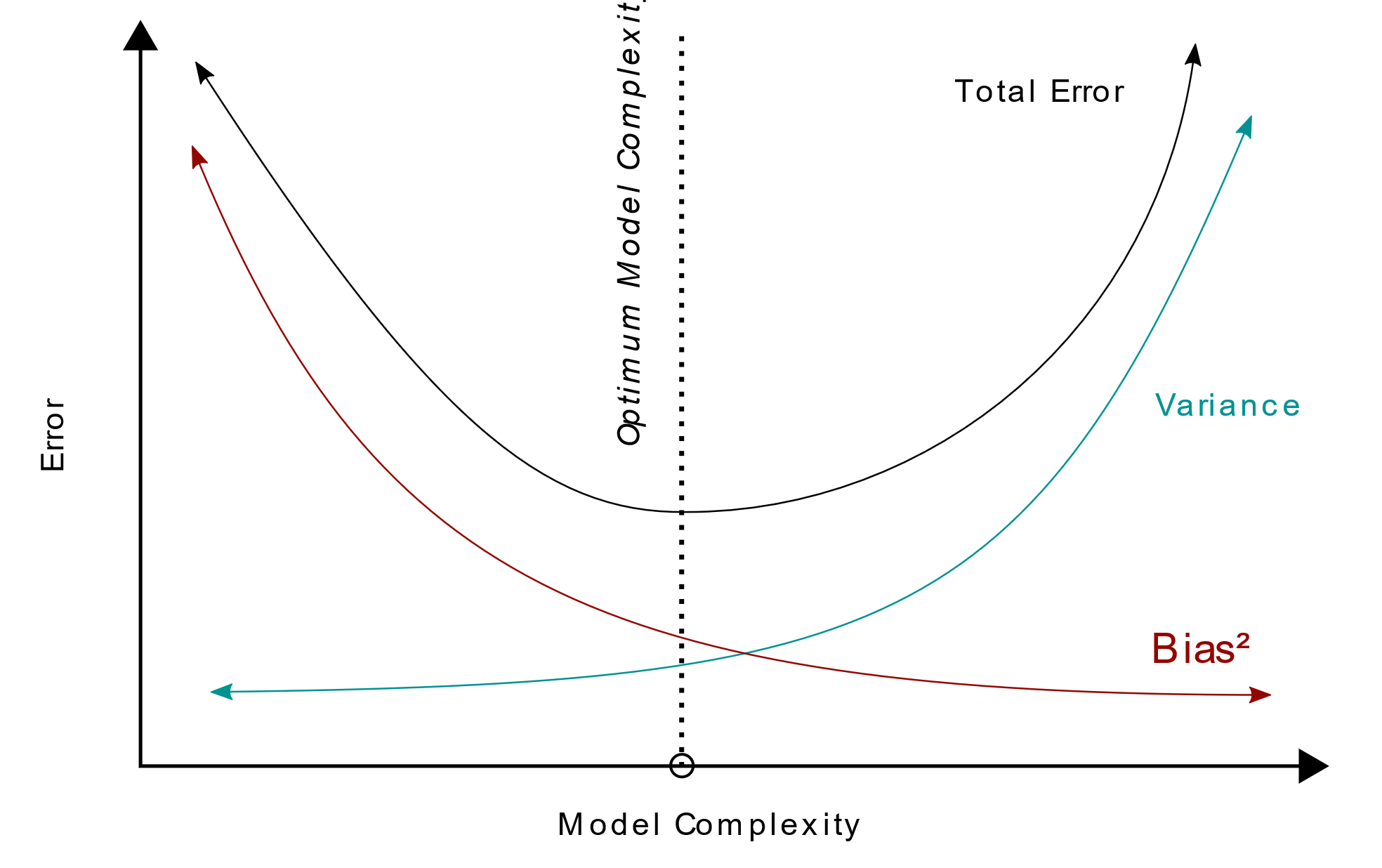


Figure 20[[12]](#footnote-12)

High bias causes the algorithm to *underfit* the problem, while high variances leads to *overfitting*. When underfit happens, the algorithm is not able to learn enough from the training data, and along with a poor performance on seen data it is not able to generalize enough to perform well of unseen data. Overfitting, instead, happens when the algorithm learns by heart the patterns in the train data, and this prevents it from generalizing well enough on new data.

In the context of Deep Learning, underfitting is generally associated with simple neural networks (few neurons and/or few hidden layers), too simple to detect meaningful patterns in the training data. Overfitting is instead associated with complex architectures, whose vast number of weights can learn to perfectly model the training data, especially when the network is trained for a long number of epochs. While underfitting can be dealt with by simply increasing the complexity of the network, avoid overfitting requires more sophisticated techniques.

The simplest way to reduce overfitting is to train the algorithm on more data, but this is not always a feasible option when dealing with real life problems. Instead, we tend to resort to other approaches, the most common being regularization, dropout and early stopping.

Regularization consists in introducing a penalty to the loss function that is generally proportional to the size of the weights in the neural network. This induces the model to shrink the value of the weights, penalizing the one that happen to be less useful when making predictions. L1 and L2 regularization, also known as Ridge and Lasso regularization, are approaches that perform such shrinking, and are expressed by the following equations:

* L1:
* L2:

In both equations, is the designated loss function and is a hyperparameter that tunes the regularization: reduces the regularization effect, while prevents the algorithm from learning.

Dropout is a technique that stimulates the model to discover “different paths” of neurons to follow when making predictions. A neural network can be seen as a directed graph consisting of hundreds of thousands of nodes, with the nodes belonging to the same layer sharing the same input connections. This makes neurons extremely dependent on each other and produces high correlation in the input features. To help the network learn several independent correlations in the data (and thus reducing the risk of overfitting), a fraction of neurons in some (or even all) hidden layer is randomly disabled during training[[13]](#footnote-13). The result is that, at each iteration, different connections among neurons are strengthened. During inference, instead, all neurons are active, and the predictions are made exploiting such different paths.

As we mentioned, overfitting is also caused by long periods of training, during which a model that is complex enough has the time to perfectly learn the most detailed particulars of the input. Since it is difficult to estimate how much time a model will need to train before incurring in the problem of overfitting (without experimenting with a different number of epochs in different runs), the idea of early stopping is to stop the training as soon as the performance of the model on unseen data (held out from the original dataset) starts to drop. After each epoch, the model is tested on such data, the performance registered, and the status of the model (values of the weights) saved. When the performance starts to drop (or after a few epochs of decreasing performance – interval called *patience*), the training is stopped, and the status of the best performing model is restored. The best model is then used during inference to make predictions on unseen data.

## Deep Learning on Graphs

The application of Deep Learning to graph-structured data has become more popular in the recent years, especially for the application in the biosciences (e.g., protein studies). The biggest challenge when dealing with such data structures is the fact that the nodes in a graph are not independent from each other, and the connection between them must be taken into account. A graph can be described by means of its adjacency matrix, and the order of its rows must not be considered as relevant. To prevent a neural network from leveraging on the order in which nodes are considered when making predictions, the operations applied to the nodes must be permutation invariant: when applied to the nodes of a graph, such operations must give the same result, independently of how nodes are arranged in the adjacency matrix.

Graph Neural Networks (GNN for short) infer information about a node by combining its features with the ones coming from its neighbors. Information are passed in form of *messages* that are collected and aggregated with permutation invariant operators[[14]](#footnote-14) by each node. If we call the features of node , the message that this node passes to its neighbors is computed as , with a learnable matrix that projects the features of node into another vector space. Once the messages from all the neighbors have been collected, the node aggregates them for example by summing them. The result is an *embedding* of the node, a vector (or more in general a tensor) that summarizes the information coming from the neighbors and from the node itself.

To compute an embedding, we must first determine the *computation graph* of each node (in Figure 22 we see the computation graph of node , constructed from the graph in Figure 21[[15]](#footnote-15)). This graph defines the flow of information coming to each node in form of messages. Since a node can only receive messages from its neighbors, the only way for it to gather information from nodes that are further than one hop is to repeat the collection and aggregation operation at each level of the computation graph. In GNN, the collection and aggregation of messages is performed by a single hidden layer, that can reach nodes that are one hop away from. To improve the scope, we stack multiple layers on top of each other. So, a GNN with two hidden layers can gather information about nodes that are two hops away, one with three layers information about nodes at three hops and so on. The number of nodes reached through the different layers represents the *receptive field* of the node.

|  |  |
| --- | --- |
| Background pattern  Description automatically generated  Figure 21 | Icon  Description automatically generated with medium confidence  Figure 22 |

Differently from what happens in neural networks used for example in image-related tasks, adding many layers is not helpful when dealing with graphs, because it can lead to the problem of *over-smoothing*, the phenomenon for which all node embeddings of a graph converge to the same value. For this reason, when working with graphs we tend to prefer *shallower* models, that is to say models with few layers, in which the number of layers is just slightly bigger than the receptive field.

Next, we will discuss two types of layers typically used in GNNs: graph attention layers, as found in Graph Attention Networks (GAT for short), and graph convolution layers, as found in Graph Convolutional Networks (GCN for short).

### GAT

On top of the message passing and aggregation of messages discussed above, Graph Attention Networks use an attention mechanism to produce embeddings of the different nodes of a graph. The attention mechanism works by computing for each node the importance that the information contained in other nodes has when building its embedding. In [16], the attention mechanism is a single-layer neural network parametrized by a weight vector , where is the dimension of the space onto which node embedding is projected after the multiplication with a learnable matrix . The attention vector is then used to perform masked attention, so to compute the attention of node only with respect to its neighbors (where is the set of all the neighbors of ). The resulting masked attention is defined for each pair of neighboring nodes and as follows:

The formula express as a normalized exponential function (*SoftMax*) that yields number between zero and one. In the formula, is the concatenation operator, and *LeakyReLU[[16]](#footnote-16)* is a nonlinear function used to increase the expressive power of the layer.

is then used to compute the new embedding of node as follows:

where is a generic nonlinear function.

In [16] is also discussed of the benefits of using a multi-head attention mechanism, that is defined for the intermediate layers as a concatenation of the embedding obtained through different attention mechanisms

and for the last layer as the average of such embeddings

### GCN

Introduced in [17], Graph Convolutional Networks are discussed with respect to the problem of classifying nodes in a graph where only a subset of labels is known (*semi­*-supervised learning problem), formulated in this way:

where denotes a supervised loss function, a neural network differentiable function and a hyperparameter that adjusts the regularization of the loss. The formulation of the problem relies on the assumption that connected nodes in the graph share similar labels (however this may restrict modeling capacity since edges in a graph do not necessarily encode node similarity).

In this framework, is defined as a multi-layer graph-based neural network, characterized by the following layer-wise propagation rule, expressed in matrix notation:

is an matrix, where is the number of nodes in the graph and the dimension of the embeddings of each node. The propagation rule describes the update of in a recursive way: is the embedding matrix after propagation steps (remember that ). In this update rule, is a learnable matrix, the adjacency matrix augmented with self-connections to each node (formally defined as , where is the identity matrix) and a diagonal matrix that contains the degree of the nodes[[17]](#footnote-17). The multiplication has the goal to normalize the terms of and avoid the problem of *exploding gradient[[18]](#footnote-18)* during training.

If we rename as , we can rewrite the propagation rule as follows:

which looks like the propagation rule of a graph attention layer in matrix notation. In this sense, a graph convolutional layer can be seen as a special case of graph attention layer with single attention where the parameters are not learnable.

In [17] is also proposed a variant of the standard GCN model which exploits residual connections[[19]](#footnote-19) between hidden layers to facilitate training on deeper models by propagating the information from the previous layer:

The idea behind residual connections is that sometimes the embeddings in previous layers of the neural network are better at describing local characteristic of the input (in this case the graph) while later embeddings better capture global characteristics. By adding the content of previous embeddings, we can increase the expressive power of the later ones.

## Deep Learning on Temporal Graphs

Dynamic graphs are a special kind of multi-relational graphs in which the nodes or the connections between them change in response to a sequence of events. Such events include the insertion or deletion of existing nodes and arcs and the change of the features of nodes and edges. When these events happen at fixed time intervals, we have Discrete Time Dynamic Graphs (or DTDG), while if the time is continuous, we have Continuous Time Dynamic Graphs (CTDG). In the following, we will refer to both DTDGs and CTDGs as time graphs when the difference in the nature of time is not important.

An important concept when dealing with time graphs is that of *temporal neighborhood*, that indicates the collection of nodes connected to a given node at a specific point (or interval) in time.

An example of dynamic graph is the one that describes a social network, where nodes and edges can appear or disappear at any time (respectively when new users open an account and others delete theirs and when they interact with each other).

### TGN

TGN, or Temporal Graph Network, is a neural network model that learns on continuous time dynamic graphs the likelihood of sequences of events [18]. In particular, the TGN model focuses on the interaction between nodes, trying to predict the next possible connection a node can establish with another node in the graph.

In this context, the dataset is composed of a sequence of arc events, and a single epoch during training goes through batches of events in chronological order. The model is trained *self-supervised*, and the future arcs are predicted using all the information from previous events: given a node , the prediction of the node it is more likely to interact with at time depends on the information collected up until time , and the accuracy of the prediction is measured by comparing it to the actual next interaction of .

The model is built according to an encoder-decoder architecture: the encoder produces temporal embeddings for the nodes, while the decoder uses these embeddings to make the predictions about future edges. It is composed of five different modules: memory, message function, message aggregator, memory updater and graph embedding module.

The memory module holds, for each node seen so far, a state (vector) that gives a compressed representation of all the past interactions of a node. It is initialized at zero and can expand to hold information about new nodes.

The message function computes the messages used to update the memory. A message includes information about the current new interaction, and for each interaction two messages are generated: one for the node initiating the interaction and one for the one receiving. If and are two nodes involved in the event , the two messages are the following:

where is the memory state associated to node at the previous time step, is the time interval occurred from the previous interaction involving the node and is additional information associated with the current interaction. and are respectively the functions that compute the messages from the source () and the one from the destination (). These functions are implemented with an MLP.

The message aggregator aggregates all the messages associated with a node in a batch (if a node appears in three different interactions inside a batch, then it will have three messages associated to it). This aggregator could consist of the mean of all the messages, or just fetch the most recent one and use it as a proxy to update the memory.

The memory updater is implemented via a *recurrent neural network*, a neural network in which the nodes are provided with a feedback loop that conditions the current output with the previous one. In this way, at each point in time, the content of the memory associated to a node contains a vector that represent the current state as the result of the evolution of the initial state of the node’s interactions through time.

Finally, the graph embedding module generates temporal embeddings of a node, based on the information collected from the nodes in its temporal neighborhood:

where is the temporal neighborhood of node during interval , a differentiable learnable function and the features of nodes and . The fact that the embedding is computed by aggregating information from its temporal neighbors solve the problem of the *staleness* of the memory: if a node has no recently interactions recorded, the information coming from its temporal neighbors can be useful, under the assumption that nearby neighbors influence each other in the evolution of the graph.

The training process is articulated in such a way to prevent problems of information leakage: first the memory is updated using the information collected from the previous batch[[20]](#footnote-20), then the temporal embeddings are computed, and the predictions made starting from such embedding. After each prediction, the information associated with the interaction (and the messages that are generated) is collected and stored in a temporary memory; its content is then retrieved before updating the real memory of the model.

# General Approach

After the discussion on the basics of Process Mining and Process Discovery in the first two chapters and the overview on Deep Learning and graphs in the last one, in this chapter we will introduce the ideas behind our. We will briefly lay out the evolution of the reasoning behind the final version of our approach to APD using Deep Learning, passing through some of the most important stages in the development of the discovery algorithm; we will then proceed to outline the main features of the algorithm, that will be discussed in more details in the next chapters.



## Difference with the State of the Art

As already mentioned, the main idea behind [1] is to develop an automatic method to discover processes that resembles the way humans would approach the problem: given the set of transitions of a Petri net, choose one place at a time by conditioning this decision on the choices made in the past. To improve on such decisions (and consider the structure of the proposed Petri net) GNNs are introduced into the algorithm: their task is to produce embeddings of the individual nodes in the graph and allow shallow MLPs to make an informed choice about which place to consider next and whether to include it in the final solution.

Our algorithm shares with [1] the use of GNNs for the embeddings and shallow MLPs to make decisions, but differs in the underlying approach to the problem: instead of emulating the human reasoning, it exploits the fact that processes develop over time, so it builds a temporal dependency among the activities in an event log by considering them in the order they appear in the trace variants. By operating on trace variants, we introduce two kinds of dependencies: one intra-variant, among the activities belonging to the same variant, and one inter-variant, linking the decisions made across the different variants of the model. For each process, the variants are sampled at each iteration during training, to prevent the algorithm from learning patterns from the order in which they are considered. This idea was what guided us in the development of the final algorithm, so it is present in all the different intermediate versions of the algorithm, though in different forms.

In all the different versions of the algorithm, the input maintained the same structure: a Petri net composed of a list of places produced according to the alpha relations of the corresponding log and the suggestions from [19] on how to build the candidate places (minus the features, that we generated in a different way).

## Most relevant failed attempts

The first idea we explored was to exploit the memory produced by the TGN to infer something about the structure of the Petri net that describes the process. In this scenario we modified the training procedure of the TGN to work on a dataset composed of different logs as opposed to a collection of chronological interactions. This included the preprocessing of the logs to turn them into lists of interactions: in the new format, a single line would not just represent a single event, but a pair of consecutive events. Two special activities, the start and end activities, were introduced in the list of interactions, so to have a clear separation between different traces. The memory of the TGN was also reset at each log to prevent information from virtually independent processes to influence one another.

A GNN-based module was added to the architecture of the TGN to make predictions about the places present in the Petri net given in input. This new model would work in two steps: first it would make the predictions about the batches of data fed into the TGN part, then the new GNN-based module would activate. This module would fetch the memory produced by the TGN part during its predictions and use this information as features for the transitions of the Petri net (while the places’ features would be initialized to zero) and then generate a prediction about the places of the input.

This module was trained in a supervised way: the input net was built on top the model produced by the Alpha miner by adding one-to-one, many-to-one, one-to-many and many-to-many places as done in [19], and so the predictions of our algorithm could be confronted with the *right* solution proposed by the Alpha miner for that event log. This approach would have had the drawback of producing solution that would have been not better that those produced by the Alpha miner. Unfortunately, this architecture proved to be unable to produced well-formed Petri nets, often yielding unconnected graph in which some transitions were isolated from the rest of the net (and therefore always enabled).

The simplest GNN-based module we tried consisted in a few GAT layers with a Sigmoid activation[[21]](#footnote-21) at the end to produce a prediction for each node. The nodes corresponding to the transitions of the Petri net were masked, so that the module could focus on the places and not produce predictions for nodes that we knew must have been in the target solution. Other solutions we explored involved more complex structures, with the GNN-based module being composed of few GAT layers and a shallow MLP to make predictions for each place individually.

Besides the already mentioned problem of not being able to produce well-formed Petri nets, such architecture was characterized by an instable loss during training, caused by the conflicting objectives of the two modules.

Other attempts were made in which the TGN was only used to preprocess the data and to produce the features of the nodes of each input Petri net. The adjacency matrix of the net, augmented with the features fetched from the TGN, was fed into a neural network composed of two parts: an encoder built using two to four layers of GAT or GCN and a decoder usually composed of a shallow MLP. Moreover, the training and inference procedure were changed: the model would not produce the predictions for the entire net in a single pass, but the procedure was repeated for each trace variant individually, and the predictions associated to the places involved in each variant were combined at the end to produce a unique and complete prediction for all the places in the net. For each trace variant, a selection of the places connected to the transitions involved was considered, and the corresponding embedding fetched from the encoder. The MLP was used to produce a log probability for each place involved in the execution of the trace variant, indicating whether a place should have been included in the result. At the end, the predictions based on the different variants were averaged according to the frequency with which each variant appeared in the log, and the averaged prediction used as result. The last layer of the MLP presented a log SoftMax[[22]](#footnote-22) activation used to produce, for each place, a prediction over two classes: class *zero* and class *one*, the first indicating that the place should not have been considered, the second suggesting the opposite. By applying the argmax operator to the result we produced a bit vector indicating whether a place should be kept in the solution.

This version of the algorithm behaved better that the previous, but still produced disconnected Petri nets, even though less frequently.

## Final approach

The final version of the algorithm adopts a more complex architecture, and a slightly different version of the training procedure described above, even though similar to what we already discussed.

The model is composed of four different modules: an encoder, which is a two-layer GCN with residual connection that produces the embedding for each node of the input graph; a module responsible for the elaboration of the initial embeddings called *info aggregator*; a decoder, that assigns to each place in the graph a score based on the elaborated features; a final module that ranks the places according to their score, chooses the best places based on that score and that computes for each place the likelihood of it being actually present in the resulting Petri net. The log likelihood of all the chosen places is the summed and used as loss during training.

Training and inference share a similar procedure. First, the data (complete of adjacency matrix and node features) is fetched and sent into the encoder, that produces the initial embedding for each node in the graph. Then these embeddings are modified by the *aggregator*. This module considers the sequence of places reached by the tokens during the execution of each trace variant and extends the embeddings of such places by including information about their surroundings in the graph. This procedure is repeated for each trace variant, and, at the end, the extended embeddings of each place are summed across the variants, so that the final embedding associated to each place is a combination of the extended embeddings produced when considering different trace variants. Let us clear this procedure with an example. Consider an event log composed of the two variants <a, b, c, e> and <a, d, c, e> and let us focus only on the places connected to activity (Figure 23) in the input Petri net (remember that the input net contains more places than needed). When reading the first variant, the *aggregator* considers the places that are reached by each activity in the variant, that in the case of activity we can call , and (in the hypothesis that has only three outgoing arcs). Their features are extended by including information about their neighbors in the graph and saved. The module proceeds to read the second variant and extend the embeddings of , and as before. When all the variants are read, the embeddings associated with are summed together, and the same happens for those of and . So, if we call the embedding of when considering the first variant and the embedding of when considering the second variant, the final elaborated embedding of will be . This procedure is done for each place reached in a variant. Beware that the neighborhood of each activity is not guaranteed to be the same across different variants, because it depends on the sequence of activities in the variant. In our example, if is connected to both activities and through place , in the first variant the neighborhood of consists of and , while in the second variant it consists of consists of and .

Graphical user interface, application

Description automatically generated

Figure 24

Once the initial embeddings are modified by the *aggregator*, they are sent to the decoder (a shallow MLP) that converts them into a number, used as a score to rank places. Then, following the breadth first order of the activities in the graph, each activity is connected to one or more places, with the guarantee of obtaining a connected graph in the end.

The difference in the procedures for training and inference lays in the fact that, during training, the log likelihood of each place is also computed (and their negative sum[[23]](#footnote-23) used as training loss) while, during inference, we are only interested in the list of places chosen for the result. Note that, by using the negative sum of the likelihoods of the nodes as a loss (that we can see as the negative log likelihood of the whole graph), our algorithm is not reliant on an annotated dataset; instead, by minimizing the negative log likelihood of the graph, the algorithm self-supervises the training procedure, learning to choose the nodes that are more useful (or likely) when describing the log. This approach is similar to the one adopted in [1], where the loss is defined as , where is the subset of available places chosen by the algorithm and the probability of choosing place after having selected places .

Training is performed on synthetic data, and the model is tested both on synthetic and real event logs.

This approach does not allow to easily add silent transitions as done in [1]. However, we added to option to extend the proposed solution with one-to-one silent transitions among the chosen places. We will see in chapter 7 how this influences the goodness of the Petri net produced algorithm.

More details about the structure of each module will be provided in chapter 0, as well as additional information about the training procedure and the inference process. In the next chapter, instead, we will focus on the data preparation, including the generation of synthetic data and the processing of real-life data.

# Data preparation

In this chapter we will examine the steps needed to build the input data for our algorithm, both from synthetic and real event logs. As we will discuss, we follow a similar approach to that indicated in [1] for what concerns the generation of synthetic logs and of the input graph. However, the input graph only consists of a bipartite graph modeling a Petri net (the component that in [1] is referenced as *candidate Petri net*). After a brief recap on the generation phase, we will illustrate the procedure followed to split the initial dataset into the train and test datasets.



## Synthetic data generation

Synthetic event logs are generated using PM4PY[[24]](#footnote-24), a Python framework for Process Mining. The first step in this process is to randomly generate a collection of process trees that exhibit some predefined characteristics. To achieve this, we use the function *apply* of the algorithm module located inside the package *tree\_generator*[[25]](#footnote-25). This function accepts a list of parameters which describes the population of the trees to generate. Since it is difficult, in general, to model OR operations and loops using Petri nets, the process trees we generate do not contain such operations. The other assumptions we make about synthetic data is that the original trees do not contain duplicate activities (i.e., activities are unique) nor silent transitions. Such behaviors are reflected in the Petri nets that we obtain by converting the operators in the trees to the equivalent structures as we showed in paragraph 1.3.5.

Once generated, each tree is converted into a Petri net[[26]](#footnote-26), which is later reduced (using a procedure that guarantees the equivalence of the original and reduced net[[27]](#footnote-27)) to get rid of the silent transitions in excess that result from the conversion. We then play out (or simulate) these Petri nets to generate the collection of event logs that forms our dataset.

## Generation of the input model

As we already mentioned, our algorithm does not take event logs directly in input, so we need to convert them into bipartite graphs, each one representing the family of models to which, in our assumption, the *correct model* (the Petri net that generated the event log) belongs. This process is performed in two steps: first, we identify the unique activities in the log (that correspond to transitions in a Petri net) and we add them to the graph; second, we derive the Alpha relations among the activities and use them to build the places to add to the graph.

In our algorithm, input graphs are modeled using the PyTroch Geometric framework, in which a *Data* objects holds, among other attributes, the node feature matrix *x* and the adjacency matrix *edge\_index*. We will discuss the node feature matrix later in this chapter; in this section we will focus on how to build the adjacency matrix.

### Transitions

Isolating the activities in a log is fairly easy: we first read the log as a set of traces exploiting the PM4PY’s function *get\_variants\_as\_tuples* in the *pm4py.stats* module, then we iterate over each trace and collect the activities into a set.

In this phase, we introduce in the set of unique activities the fictitious activities > and |, that represent respectively the starting and ending activities of each trace in the log, so that we can construct input graphs that have a unique starting and ending. We then transform them into silent transitions during inference, to prevent them from interfering with the actual behavior of the net.

|  |  |
| --- | --- |
| **Algorithm 5.1** Find unique activities | |
|  | **Input:** set of variants V |
|  | **Output**: set of unique activities A |
|  |  |
| 1: |  |
| 2: | **for**  **do**: |
| 3: | **for**  **do**: |
| 4: |  |
| 5: | **end for** |
| 6: | **end for** |
| 7: |  |
| 8: |  |
| 9: | return |

Since the activities in an event log are usually represented by strings, we need to map them to integer values and use such numbers to build the adjacency matrix. Such map is built as follows:

|  |  |
| --- | --- |
| **Algorithm 5.2** Activity map | |
|  | **Input:** set of unique activities A |
|  | **Output**: map M associating activities to integers |
|  |  |
| 1: |  |
| 2: |  |
| 3: | value = 1 |
| 4: | **for**  **do**: |
| 5: |  |
| 6: | value = value + 1 |
| 7: | **end for** |
| 8: |  |
| 9: | return |

### Places

Places are constructed starting from the footprint of the corresponding log. Similar to what is described in [19], we generate places that correspond to one-to-one, one-to-many, many-to-one and many-to-many relations among activities. All relations are based on the Alpha relations among the activities in the log: one-to-one places correspond to places that connect activities that are directly followed by one another in the log, one-to-many places correspond to places that connect one activity to at least two other activities that directly follow it and so on. One-to-one places also model parallel relations among activities, and this happens when one activity is the source for more than one one-to-one places (Figure 25). One-to-many, many-to-one and many-to-many places, instead, model choices among activities.

During this phase, places are described as pairs of source and target activities. For instance, if place connects activity to activities and , we represent it as the tuple . Only when all places are generated the tuples representing them are converted into actual nodes.

Graphical user interface, application

Description automatically generated

Figure 26

We mentioned that during the preprocessing we add the fictitious activities *>* and *|*. These two activities are connected, respectively, to the source and sink places of the graph, and from them stem the places that fire or consume the tokens for the execution of the actual starting and ending activities. An assumption we made is that a process does not start (nor ends) with parallel activities. For this reason, the places that connect > and | to the activities of the process are, respectively, a one-to-many place and a many-to-one place (so starting and ending activities are always in choice relation with each other). The creation of such places is taken care after the creation of the places based on the footprint of the log (that does not include > and |).

The following is the general algorithm that generates all the places of the input graph. The functions called during the execution will be described in the next sections of this chapter.

|  |  |
| --- | --- |
| **Algorithm 5.3** Generate places | |
|  | **Input:** set of unique activities A, footprint F, activity map M |
|  | **Output:** set of places P, node map M |
|  |  |
| 1: |  |
| 2: |  |
| 3: |  |
| 4: |  |
| 5: |  |
| 6: |  |
| 7: |  |
| 8: | **for** **do:** |
| 9: | **if** **then:** // if a is a starting activity |
| 10: |  |
| 11: | **end if** |
| 12: | **if** **then:** // if a is an ending activity |
| 13: |  |
| 14: | **end if** |
| 15: | **end for** |
| 16: |  |
| 17: |  |
| 18: |  |
| 19: |  |
| 20: | return P, M |

Like we did for the activities, we need to map the name of the places to integer values in order to build the adjacency matrix for the resulting graph. The mapping is similar to the one used for the activities; in fact, we extend the activity map by numbering places starting from value (where is the original activity map) and we consider the places in the order they were generated. Lines 2 to 5 in the above algorithm show how each function returns the updated set of places (we assume that the order in which places are introduced is maintained) and the updated map.

#### One-to-one places

One-to-one places are the easiest to construct: starting from the directly-follows relations among activities, we create a place for each pair of source-target activities and encode it as a tuple. For example, given the log , we would generate places , , and that connect activities that directly follow one another. The following is an example of one-to-one place:



Figure 27

#### One-to-many and many-to-many places

One-to-many places are obtained by associating each activity with all the possible combinations of its (direct) followers. From this collection of combinations, we exclude those containing activities that are parallel to each other. Let us consider the example in which activities are in a choice relation after the execution of activity (meaning, after executing we can only execute one activity among ). In this case, the one-to-many places that we would generate are the possible combinations (of size two and above) of choices we can make about : , , and . If instead, in a similar setting, and , we would only generate the place .

The algorithm that produces such places is the following:

|  |  |
| --- | --- |
| **Algorithm 5.4** One-to-many places | |
|  | **Input:** set of places P, set of unique activities A, footprint of the log F |
|  | **Output:** set of places P |
|  |  |
| 1: |  |
| 2: | **for do:** |
| 3: | // get the activities that directly follow *a* and that are not parallel to each other |
| 4: |  |
| 5: | **for do:** |
| 6: |  |
| 7: | **end for** |
| 8: | **for**  **do:** |
| 9: | **if**  **then:** |
| 10: |  |
| 11: |  |
| 12: |  |
| 13: | **end if** |
| 9: | **end for** |
| 10: | **end for** |
| 11: | return P |

In the algorithm, returns the combinations of size of the elements in .

The procedure to generate many-to-one places is similar to the one just described, with the difference that we generate the combinations of the activities that directly precede a given activity.

In Figure 28 and Figure 29 we can see, respectively, an example of one-to-many places and many-to-one-places.

|  |  |
| --- | --- |
| Figure 28 | Graphical user interface, application  Description automatically generated  Figure 29 |

#### Many-to-many places

For the creation of many-to-many places, we need to consider all the combinations of pairs of subsets of the set of activities that do not contain parallel activities.

Many-to-many places have multiple incoming and outgoing transitions that can be represented as subsets of the set of all the activities in the process. The first step is therefore to generate all the proper subsets of the set of activities, and while doing so we filter out all the subsets that contain pairs of parallel activities. In mathematical terms:

Then we list all the possible pairs of such subsets, and we check that in each pair the activities in the first set are directly followed by the activities in the second set. The pairs that do not satisfy this condition are excluded from the list. Those that remain are all the feasible many-to-many places that we can put in the input graph. In mathematical terms:

In Figure 30 we can see an example of a many-to-many place.

Graphical user interface, application, Teams

Description automatically generated

Figure 30

### Bringing it all together

Once places and activities/transitions are ready, we can put them together and build the input graph, remembering that we need to convert the names of the nodes using the node map that we created during the generation of transitions and places. We obtain the (sparse) adjacency matrix for the input graph with the following algorithm:

|  |  |
| --- | --- |
| **Algorithm 5.5 Adjacency matrix** | |
|  | **Input:** set of places P, node map M |
|  | **Output:** (sparse) adjacency matrix Adj |
|  |  |
| 1: |  |
| 2: |  |
| 3: | **for do:** |
| 4: |  |
| 5: | **for**  **do:** |
| 6: |  |
| 7: | **end for** |
| 8: | **for**  **do:** |
| 9: |  |
| 9: | **end for** |
| 10: | **end for** |
| 11: | return Adj |

The sparse matrix is then converted into a 2D tensor as required by PyTorch Geometric. In the attempt to allow nodes to pass messages to their parents (incoming nodes), during this conversion all the connections are made bidirectional.

## Node features

We have explored three different options for the features of the nodes, all independent from the number of the activities in the log.

As mentioned, in the initial stages of the development of the algorithm we used the memory of the TGN as source of information for the features of the nodes corresponding to the activities in an event log, while the features of the nodes corresponding to the places were initialized to zero. This approach can be used also in the last version of this algorithm, even though, being the two models separated, it requires to run the TGN on each log in the dataset and save the updated memory each time. Each memory should then be fetched and its cells used as features of the corresponding activity nodes in the input graph. This kind of feature embeds temporal information about the activities in the log but does not provide any hint about the structure of the process.

Another approach is to generate random features from a gaussian distribution. While this works in practice, it has the drawback of not considering any extra information about the log that could be helpful during training or inference.

The last approach we investigated consisted in creating features based on the temporal DFG associated with the log (in this case augmented with the introduction of > and | at the beginning and end of each trace). In this method, the idea is to produce, for each activity, an embedding based on its neighborhood at different timestamps. This temporal neighbor is defined within a time-window centered around an activity at a given time. By averaging the embeddings of a node at different times, we obtain a single embedding that has the same size of the time window we used. Also in this case, the features of the places are initialized to zero. The following is the algorithm we used to produce the temporal embedding of a single activity node, in which we assume that gives the neighbors (annotated with their timestamp) of node within a time interval:

|  |  |
| --- | --- |
| **Algorithm 5.6 Temporal features** | |
|  | **Input:** node , set T of timestamps at which is executed, set of temporal neighbors , window *w*, node map M |
|  | **Output:** temporal embedding *e* of node *n* |
|  |  |
| 1: |  |
| 2: | **for do:** |
| 3: |  |
| 4: |  |
| 5: |  |
| 6: | **for do:** |
| 7: |  |
| 8: | **end for** |
| 9: | **for do:** |
| 10: |  |
| 11: |  |
| 12: | **end for** |
| 13: |  |
| 14: | **end for** |
| 15: | return *e* |

Regardless of how they are produced, the features of the nodes in an input graph are first normalized and then stacked to form an matrix, where is the number of nodes in the graph and the size of the features.

## Train/test split

Since the training dataset is generated randomly, and since the footprint of each log indirectly determines the number of places in the corresponding input graph, the population we generate contains graphs that have different number of nodes and different mean node degree. When we partition the dataset to obtain a test set, we need to sample subsets that are representative of the characteristics of the original population. For this reason, we choose to split the dataset into train and test sets using stratified sampling.

To identify the different strata of the population, we need to classify the graphs based on their characteristics. For each input graph we compute the number of nodes, the mean node degree and the standard deviation in the degree of its nodes and then we use an unsupervised method to cluster the dataset based on this information. First, we discretize such variables using the KBinDiscretizer module available in Scikit learn[[28]](#footnote-28) with *quantile* strategy, that assigns each variable to a *bin* with the objectives to produce bins that have approximately the same number of samples. In our case, we decided five bins for each variable were enough. Then we cluster the graphs using pandas[[29]](#footnote-29) by producing a data frame split into groups (our strata). Each stratum is then sampled using random sampling to produce a partition in which the train set contains 80% of the population and the test set the remaining 20%.

# Architecture, training, and inference

In this chapter we will delve into the description of the working of our approach, starting from the explanation of its architecture. After that we will analyze the training procedure, with a qualitative analysis of the synthetic dataset used during this phase, and the inference procedure, with a brief digression on the extension of the solutions proposed by our approach with the introduction of silent transitions.

## Architecture

Our discovery algorithm is built around a neural network-based model composed of four main modules: an encoder, a decoder, a module that elaborates the initial features of the input that we will call *information aggregator* and a module that decides which places of the input should be present in the result. Figure 31 illustrate the dependency among the different modules, with the arrows also outlining the information flow inside the model.

Diagram

Description automatically generated

Figure 31

### Encoder

The purpose of this module is to produce an embedding of the nodes of the input graph starting from their initial features. It consists of a shallow two-layered GNN with skip connections implemented using graph convolutional layers. With reference to equation 16 that we have presented in paragraph 3.3.2, this module can be described with the following equations:

where is the initial feature matrix (each row is the feature associated with a node) and is the activation function.

The encoder is used at the beginning, to obtain embeddings starting from the initial features of the nodes, and later in the *information aggregator*, as we will discuss in the following paragraph.

### Information aggregator

This is the most complex module in the model and has the core objective to learn the embeddings that will be used later to choose which places should be added to the resulting Petri net.

As briefly introduced in paragraph 4.3, this module tries to condense into a single embedding the information associated with the neighborhood of each place in the context of different variants, i.e., different executions of the process. The *information aggregator* takes the initial embeddings produced by the encoder and, for each variant in the process’s event log, it produces the list of places that could be involved in the execution of the process, taken in the order in which they appear in the execution; following this order it then produces new embeddings for each place. The new embeddings are constructed as follows: for each place, we take its source and destination transitions (according to the variant we are considering) and we concatenate the initial place’s embedding to the embeddings of these transitions. Let us consider the example in which place is connected to transition (source) and (destination); their embedding are coming from the encoder, so they all have the same shape (Figure 32). The new embedding of place is obtained by concatenating such embeddings, obtaining a tensor that is three times the size of the original ones (Figure 33). This operation is repeated for each place involved in the execution of the variant. Except for the first place encountered in the execution, the embedding of each place is summed with the embeddings of the places previously considered during this procedure, forwarding the information of the execution to the places involved in later steps. In mathematical terms:

where and are obtained by the concatenation previously described, and is the resulting embedding of each place beside the first one encountered in the execution.

|  |
| --- |
| Shape  Description automatically generated with medium confidence  Figure 32 |
| Figure 33 |

This procedure is repeated for each variant in the event log, and the embeddings found are stored for later use.

Once all the variants have been considered, each place is associated with embeddings, where is the number of variants in the event log. At this point, each embedding only contains partial information about the behavior of the entire process, so we cannot use them directly to make informed (global) decisions about the presence of the places in the final Petri net. Instead, we first need to aggregate them so that each place’s embedding summarizes the context of its involvement in the execution across different variants. For this reason, we sum the embeddings of each place, as shown in Figure 34.

Background pattern

Description automatically generated

Figure 34

The algorithm that describes this module is the following:

|  |  |
| --- | --- |
| **Algorithm 6.1** Information aggregator | |
|  | **Input:** initial embeddings , adjacency matrix , list of variants |
|  | **Output:** final embeddings |
|  |  |
| 1: |  |
| 2: |  |
| 3: | **for** **do:** |
| 4: |  |
| 5: | **for** **do:** |
| 6: |  |
| 7: | **for**  **do:** |
| 8: |  |
| 9: |  |
| 10: |  |
| 11: |  |
| 12: |  |
| 13: | **end for** |
| 14: |  |
| 15: |  |
| 16: | **end for** |
| 17: |  |
| 18: | **end for** |
| 19: | return |

Matrices and in the algorithm represent respectively the matrix of the initial embeddings and the matrix of the final embeddings. The former is of shape , while the latter is of shape , where is the number of nodes and is the dimension of the initial embedding. Note that is initialized to zero, and at the end of the algorithm all the rows corresponding to transition nodes are unchanged.

As we can see in line 15, this module uses the encoder we described in the previous paragraph to update the initial embeddings after the execution of each activity in a variant. This is equivalent to add new layers to the encoder, with the difference that the number of layers is not the same for all the nodes but depends on the distance of each node from the starting activity of the variant (i.e., the distance in the minimum spanning tree constructed on the Petri net starting from activity <). At the end of the loop that began on line 5, the matrix of the initial embeddings is restored to its initial value, to avoid over-smoothing.

### Decoder

The decoder is a simple four-layered MLP with a LeakyReLU activation function between each hidden layer. It takes in input the embeddings produced by the *information aggregator* and returns a single output value for each embedding. This value is the score used by the *chooser* module to rank the places and select the ones that will end up in the proposed Petri net.

### Chooser

This module selects the places that will be added to the final Petri net. Given an activity, it uses the decoder to rank the places that are in the forward star of the corresponding transition node and selects the top places so that the directly follows constraints on the activity considered are satisfied. This ensures that the resulting Petri net is connected. It also computes the log likelihood of the chosen places, that are then summed and used to compute the loss during training.

## Core algorithm

We have discussed how the different modules operate, what is their input and what is their output. Now we will see how these modules interact to produce a Petri net given an input graph. The following algorithm is the equivalent of the *forward* method of the model’s implementation in PyTorch.

|  |  |
| --- | --- |
| **Algorithm 6.2** Forward method | |
|  | **Input:** initial features , adjacency matrix , list of variants , list of nodes |
|  | **Output:** list of places |
|  |  |
| 1: |  |
| 2: |  |
| 3: |  |
| 4: |  |
| 4: | **for do:** |
| 5: | **for** **do:** |
| 6: |  |
| 7: |  |
| 8: | **end for** |
| 9: | **end for** |
| 10: | **return** |

The set that is returned by the algorithm is composed of tuples like those described in paragraph 5.2.2. Once the places are discovered, the list of places and the one of the transitions are merged and converted into an actual Petri net following the constraints expressed in the adjacency matrix.

## Training

### Synthetic dataset

### Training procedure

The model composed of all the modules described is trained in a supervised manner, with the objective of minimizing the following function:

where is the set of chosen places for a given input and the value produced by the *decoder* module. The backward pass on the modules of the model is performed for a variable number of times for each input sample. This repetition is to allow the model to catch up the inter-variant relationships in the nodes of the input graph, and therefore learn to build better embeddings for each node. The loop is executed until one of two conditions is satisfied: either the difference in the negative log likelihood of two consecutive solutions is within a predefined or the maximum number of allowed iterations is reached. The loss at each iteration is averaged, and this value summed with the average loss of all the other samples. This sum of average losses is used as indicator to interrupt the training procedure according to an early stopping criterion: if this value does not decrease for epochs (the *patience* of the training method) the training is interrupted, and the weights of the model are restored to those associated with the epoch for which the sum of average losses is the lowest.

|  |  |
| --- | --- |
| **Algorithm 6.3** Mock-up of the training procedure | |
|  |  |
| 1: | **for** **do:** |
| 2: | // instructions |
| 3: | **for** **do:** |
| 4: | // instructions |
| 5: | **while do:** |
| 6: | // instructions |
| 7: | **end while** |
| 8: | **end for** |
| 9: | **end for** |

## Inference

The inference procedure simply consists in a call of algorithm 6.2, without the need of computing the negative log likelihood of the produced Petri net. This results in a Petri net with only two silent transitions, the ones associated with activities < and |. As mentioned before, the way this algorithm is written does not allow for an easy introduction of silent transitions during the discovery, limitation that impacts on the way the messages are distributed among the nodes by the encoder. However, we implemented a procedure to extend the produced Petri net with silent transitions among the chosen places to increase the expressive power of the solution.

### Silent transitions

# Results



## On synthetic data

## On real-life data

# Conclusion

# Bibliography

|  |  |
| --- | --- |
| [1] | V. M. D. F. Dominuque Sommers, "Process Discovery Using Graph Neural Networks," 2021. |
| [2] | W. v. d. Aalst, Process Mining: Data Science in Action, Springer, 2016. |
| [3] | W. v. d. Aalst, "Business Process Management: A Comprehensive Survey," 2012. |
| [4] | M. Tadao, "Petri Nets: Properties, Analysis and Applications," 1989. |
| [5] | W. v. d. Allst, "Causal Nets: A Modeling Language Tailored towards Process Discovery," 2011. |
| [6] | A. Berti and W. v. d. Aalst, "Reviving Token-based Replay: Increasing Speed While Improving Diagnostics," in *ATAED@ Petri Nets/ACSD*, 2019. |
| [7] | B. v. Dongen, J. Carmona and T. Chatain, "Alignment-based Metrics in Conformance Checking". |
| [8] | J. Dehnert and P. Rittgen, "Relaxed Soundness of Business Processes," 2001. |
| [9] | J. Muñoz and G. J. Carmona, "A fresh look at precision in process conformance," in *International Conference on Business Process Management*, Berlin, Heidelberg, 2010. |
| [10] | A. Adriansyah and e. al, "Measuring precision of modeled behavior," *nformation systems and e-Business Management 13.1,* 2015. |
| [11] | F. R. Blum, "Metrics in process discovery," Computer Science Department, University of Chile, 2015. |
| [12] | J. C. Buijs, B. F. v. Dongen and W. v. d. Aalst, "Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity," *International Journal of Cooperative Information Systems 23.01,* 2014. |
| [13] | A. Weijters, W. v. d. Aalst and A. K. A. D. Medeiros, "Process mining with the HeuristicsMiner algorithm," 2006. |
| [14] | S. Leemans, D. Fahland and W. v. d. Aalst, "Discovering Block-Structured Process Models From Event Logs - A Constructive Approach". |
| [15] | A. Augusto, R. Conforti, M. Dumas, M. L. Rosa, F. Maggi, A. Marrella, M. Mecella and A. Soo, "Automated Discovery of Process Models from Event Logs: Review and Benchmark," *IEEE Transaction son Knowledge and Data Engineering,* vol. 31, 2019. |
| [16] | P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò and Y. Bengio, "Graph Attention Networks". |
| [17] | T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks". |
| [18] | E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti and M. Bronstein, "Temporal Graph Networks for Deep Learning on Dynamic Graphs". |
| [19] | D. Sommers, Process discovery using machine learning, 2020. |
| [20] | T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE,* pp. 541-580, 1989. |

# List of Figures

[Figure 1 11](file:///C:\Users\matte\Documents\GitHub\GLPD\documents\thesis_10499057.docx#_Toc113029418)

[Figure 2 13](file:///C:\Users\matte\Documents\GitHub\GLPD\documents\thesis_10499057.docx#_Toc113029419)

[Figure 3 14](#_Toc113029420)

[Figure 4 15](file:///C:\Users\matte\Documents\GitHub\GLPD\documents\thesis_10499057.docx#_Toc113029421)

[Figure 5 15](file:///C:\Users\matte\Documents\GitHub\GLPD\documents\thesis_10499057.docx#_Toc113029422)

[Figure 6 18](#_Toc113029423)

[Figure 7 18](#_Toc113029424)

[Figure 8 20](#_Toc113029425)

[Figure 9 21](#_Toc113029426)

[Figure 10 27](#_Toc113029427)

[Figure 11 27](#_Toc113029428)

[Figure 12 30](#_Toc113029429)

[Figure 13 30](#_Toc113029430)

[Figure 14 32](#_Toc113029431)

1. The discipline that focuses on process modeling and the implementation of processes (not necessarily in a data driven way). [↑](#footnote-ref-1)
2. Such markings are called dead markings. [↑](#footnote-ref-2)
3. The description given are based on the resources found on the website of the Process Mining framework called PM4PY (<https://pm4py.fit.fraunhofer.de/>), since this is the framework used to assess the quality of the project which is the main content of this document. [↑](#footnote-ref-3)
4. In computer science, it is a measure of the dissimilarity of two strings, computed by counting the minimum number of operations required to transform one string into the other. [↑](#footnote-ref-4)
5. Remember that in a transition system, the arcs connecting the states are labeled with the name of the activity that, if executed, allows to transit from a state to another. [↑](#footnote-ref-5)
6. Example taken from lesson 10 “Inductive Miner” of the course “Process Mining Course @ RWTH Aachen University (BPI 2021)” available on YouTube. [↑](#footnote-ref-6)
7. Split miner, Heuristic miner, ILP miner and Inductive miner. [↑](#footnote-ref-7)
8. In simple graphs, since the only important information to retain is the presence (one) or absence (zero) of a link between nodes. In other types of graphs, especially if we want to express the strength (or weakness) of a link, (or other multi-dimensional sets). [↑](#footnote-ref-8)
9. Even though they are two different concepts, in the following chapter we will use the two terms interchangeably when such different is not important to the idea we are trying to express. [↑](#footnote-ref-9)
10. Neurons that share the same inputs and outputs. [↑](#footnote-ref-10)
11. If we assume to have a composite function , then the derivative of *y* with respect to *x* is [↑](#footnote-ref-11)
12. From Wikipedia’s article “Bias-variance tradeoff” ([Bias–variance tradeoff - Wikipedia](https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff)). [↑](#footnote-ref-12)
13. This happens for each layer independently. [↑](#footnote-ref-13)
14. Sum, min, max and mean are examples of permutation-invariant operators. [↑](#footnote-ref-14)
15. Example taken from lesson 6.3 “Deep Learning for Graphs” of the course “CS224 – Machine Learning with Graphs” available on YouTube. [↑](#footnote-ref-15)
16. [↑](#footnote-ref-16)
17. The sum of the number of incoming and outgoing arcs of a node. [↑](#footnote-ref-17)
18. Big values of the gradient of the error during training can lead to substantial increments in the value of the weights of a neural network, making the training process unstable. [↑](#footnote-ref-18)
19. Consider an input that is fed to a layer that learns the desired mapping . A residual connection is a kind of skip-connection that allows the layer to instead learn another mapping , called residual. The original mapping is therefore expressed as . [↑](#footnote-ref-19)
20. Since the first batch of data does not have a predecessor, the memory is initialized to zero. [↑](#footnote-ref-20)
21. S-shaped function that produces an output that is between zero and one, usually interpreted as the probability of the event described by x (e.g., in classification problems). [↑](#footnote-ref-21)
22. , with the number of classes considered. It is also called *normalized exponential function*. [↑](#footnote-ref-22)
23. , where is the likelihood of place . [↑](#footnote-ref-23)
24. Version 2.2.29 [↑](#footnote-ref-24)
25. pm4py.algo.simulation.tree\_generator [↑](#footnote-ref-25)
26. Function *apply* of the module *to\_petri\_net* inside package *pm4py.objects.conversion.process\_tree.variants* [↑](#footnote-ref-26)
27. We use PM4PY’s implementation of Murata’s six rules for Petri net reduction [20]. [↑](#footnote-ref-27)
28. https://scikit-learn.org [↑](#footnote-ref-28)
29. https://pandas.pydata.org [↑](#footnote-ref-29)