Technische Universität Berlin

A Deep Learning and NLP-Based Approach for Trace-Forecasting in Predictive Process Mining

by

Maximilian Oliver Fisch

Matriculation Number 398153

A thesis submitted to

Technische Universität Berlin
School IV - Electrical Engineering and Computer Science
Department of Telecommunication Systems
Service-centric Networking

Bachelor's Thesis

December 13, 2023

Supervised by:
Prof. Dr. Axel Küpper

Assistant supervisor:
Prof. Dr.-Ing. Sebastian Möller

SNET
SERVICE-CENTRIC NETWORKING

Eidestattliche Erklärung / Statutory Declaration

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed.

Berlin, December 13, 2023 Maximilian Oliver Fisch

Abstract

The pursuit of process optimisation, a fundamental component of Process Mining, fuels the persistent pursuit of innovative methodologies. This quest extends to the realm of enhancing processes during runtime, where Predictive Process Mining plays a pivotal role. Concurrently, the emergence of Deep Learning, exemplified by transformative models like Transformers, has transcended its initial success in Natural Language Processing tasks to find applications in diverse domains. This thesis proceeds through the historical evolution of Deep Learning approaches, intricately intertwined with the progress in Predictive Process Mining. As the demand for real-time process improvement grows, the exploration of cutting-edge technologies becomes imperative. Motivated by the concept of abstracting an event log into a token-based "Process Language," this thesis introduces a decoder-only transformer-based model. The model, equipped with an integrated tokeniser, aligns with its core objective, which is to forecast the continuation of a trace within a given XES-event-log. The approach will not only be conceptualises, but also evaluates it, demonstrating that it requires further modifications to fulfill its intended purpose. The model's foundation lies in the preprocessing of XES event logs, where translation maps and abstracted logs are generated. These abstractions serve as inputs to the model, supposedly enabling it to predict how a trace is likely to unfold. In essence, this thesis stands at the crossroads of (Predictive) Process Mining, Deep Learning, and Process Optimisation.

Zusammenfassung

Das Streben nach Prozessoptimierung, ein grundlegender Bestandteil des Process Mining, fördert das Bestreben nach innovativen Methoden. Diese Suche erstreckt sich auch auf Bereiche wie der Verbesserung von Prozessen zur Laufzeit, ein Themengebiet, in dem Predictive Process Mining eine wesentliche Rolle spielt. Unabhängig davon hat die Anwendung von Modellen des Deep Learning, wie z.B. Transformer, auch abseits ihrer ursprünglichen Domänen wie Natural Language Processing, in verschiedensten Bereichen zu Fortschritten geführt. In dieser Bachelorarbeit wird die historische Entwicklung des Deep Learning und der damit verbundene Fortschritt im Predictive Process Mining betrachtet. Mit dem steigendem Interesse an Echtzeit Einblicken und der Möglichkeit zur Prozessoptimierung zur Laufzeit, wird das Ausloten neuester Technologien unumgänglich. Aufbauend auf dem Konzept, Prozessausführungsdaten in eine Token-basierten "Prozess-Sprache" zu abstrahieren, stellt diese Arbeit eine auf Transformer basierende Dekoder-Architektur vor. Um das Ziel, den Fortlauf von Prozesssequenzen durch das Auslesen und Verarbeiten von XES-Event-Logs vorhersagen zu können, ist ein Token-Ersteller in das Model integriert. Dieses Vorgehen und das Modell werden in der folgenden Arbeit sowohl konzeptionell erstellt, als auch anschließend evaluiert. Die Evaluation des Modells zeigte deutlich, dass Änderungen und weitere Optimierungen der Architektur notwendig sind, damit das Modell die zuvor beschriebene Aufgabe wie beabsichtigt erfüllt. Das Fundament des Modells stellt dabei die Vorbearbeitung der Prozessdaten dar. Bei dieser werden Übersetzungskarten, als auch Datenabstraktionen der ursprünglichen Prozessdaten erstellt. Diese Prozessdaten werden dem Modell als Eingabedaten übergeben, welches damit Voraussagen über den wahrsscheinlichen Verlauf des Prozesses prognostiziert. Thematisch ist diese Arbeit in der Schnittmenge von (Predictive) Process Mining, Deep Learning und Prozessoptimierung einzuordnen.

Contents

1 Introduction

Regardless of the factors in question, such as time, cost, efficiency, or any other aspect, the analysis and optimisation of processes has been and continues to be a persistent and perpetual requirement in various domains. The historical roots of this phenomenon are evident in numerous multifaceted perspectives and methodologies employed and embraced in a wide range of disciplines. Scientific Management (e.g., Taylorism [1]), Political Economy (Adam Smith [2, 3]), or methodologies like Total Quality Management [4], Business Process Reengineering [5], and Lean Manufacturing [6, 7] are just a few of many in an ever-growing array of theories and approaches orbiting the realm of processes. These diverse theories encapsulate aspects ranging from efficiency and quality to cost reduction and customer satisfaction, each addressing different facets evolving around the process.

The rationale behind such considerations may differ, for instance, a government agency providing administrative services to citizens may benefit from digitisation, as it can streamline processes, eliminate waste, such as waiting time, and thus enhance overall efficiency. The digitisation could therefore enhance the overall perception of both workers and customers, while also reducing unnecessary expenses. In the context of a car manufacturing process, determining and enhancing the duration required for a yield could be of significant interest, as its minimisation would result in a rise in throughput and consequently, an increase in production. Adopting a systematic methodology, applying robust and proven optimisation, rooted in scientific principles, can be instrumental in quantifying and improving the temporal and financial dimensions of the process. Unsurprisingly, the need for such steady and enduring approaches and views to and onto processes begets and fosters the volition in the scientific community to continue research in the general domain of Business Process Management (BPM), as the linchpin for essentially every possible adaption of a process

Moreover, given the advancement of technology, as exemplified by theories such as Moore's Law [8] (observing the doubling of transistors in a microchip every two years) or Kryder's Law [9] (tracking the increase in hard disk drive storage density), the involvement in process advancements appears to be a crucial factor in a world driven by economics. This development, in turn, aligns with the overall notion that any process is susceptible to improvement over time, while also highlighting that crafting a process that is impervious to enhancement, particularly within the realm of Information Technology (IT), is improbable if not impossible. As these laws and theorems appear to be substantiated by empirical evidence [8], it further indicates that fundamental technological progress is accompanied by continuous prospects for process refinement and optimisation.

The unceasing need for optimisation emerges as a pivotal determinant. As industries evolve, adapt, and integrate technological advancements, the pursuit of efficiency and effectiveness becomes not only a strategic advantage but a prerequisite for economic survival. This introduction sets the stage for a comprehensive exploration of process optimisation, laying the foundation for the subsequent examination of Deep Learning (DL) and Natural Language Processing

(NLP) in the realm of Predictive Process Mining (PPM) to enhance processes by predictive insights in its dynamics.

1.1 Business-Process-Lifecycle

Before delving into the subject of Process Mining (PM), we provide a succinct overview of the present state of the art in the domain of Business Process Management to provide a brief overview of the background.

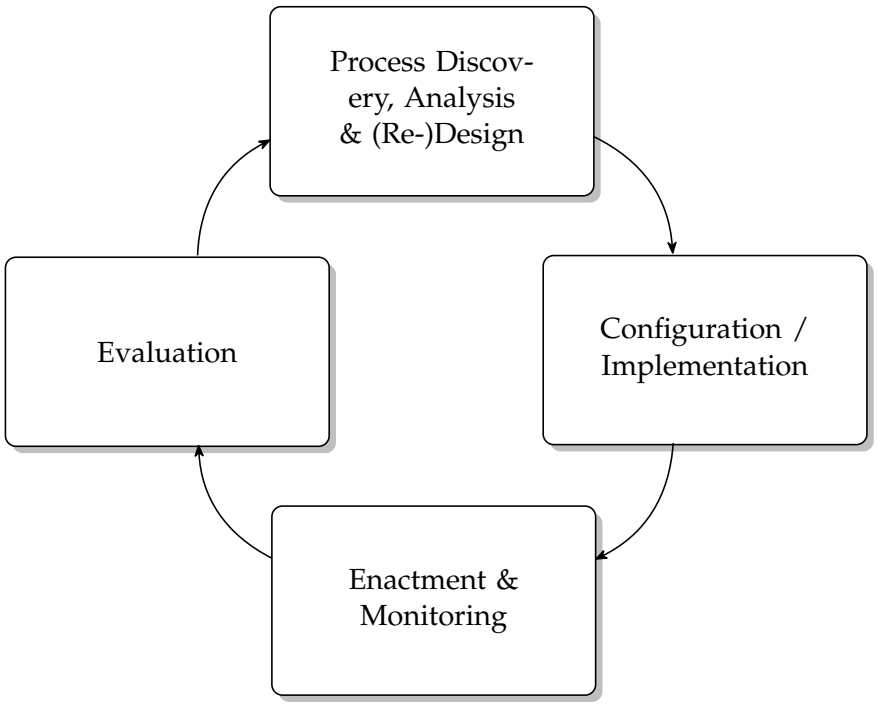


Figure 1.1: BPM lifecycle (adapted after: [10, 11]).

BPM serves as the conceptual framework for choreographing, orchestrating and optimising organisational processes [10, 11]. The BPM lifecycle, as depicted in Figure 1.1, which has been adapted following the work of Weske et al. [10] and Dumas et al. [11], encompasses the iterative progression of processes throughout their lifespan, thereby providing a structured approach for their management.

- **Process Discovery** entails the identification and comprehension of existing processes within an organisation; a step that often evolves around modelling or remodelling of the process (Process (Re-)Design). Through techniques like interviews, documentation analysis, and workshops, the goal is to uncover the intricacies of workflows [10, 11]. This could also be achieved using PM techniques, such as the Alpha Algorithm or the heuristic mining approach [12]. The step also involves **Process Analysis**, i.e. validation, simulation, and verification [10], to identify bottlenecks or evaluate the process based on metrics. Process Discovery also allows for the subsequent **(Re-)Desing** of the process in

an optimised model that considers all found opportunities for improvement.

- Following the analytical phase, the **Configuration / Implementation** phase focuses on transforming the refined process designs into operational realities, i.e. transitioning from a hypothetical model to an operational process model. This entails the system selection, implementation, testing, deployment, and configuration of appropriate tools and technologies to facilitate the execution of processes [10]. Successful Configuration and Implementation ensures that the designed processes can be seamlessly integrated into the organisation's workflow, ready for enactment.

- The **Enactment & Monitoring** phase marks the execution, monitoring, and the maintenance of the configured processes in the live operational environment [10]. It involves the continuous monitoring of process performance during its execution, e.g. based on relevant metrics. Monitoring allows organisations to identify deviations from the expected norms, which may enable timely intervention to address issues and optimise ongoing processes. It also collects the necessary data for the following evaluation.

- The **Evaluation** involves the systematic assessment of the enacted processes against predefined performance criteria. Evaluation serves as a feedback loop, providing valuable insights into the conformance and performance of the implemented processes and highlighting areas for further improvement [11]. Lessons learned during this stage feed back into the Process Discovery, Analysis & (Re-)Design phase, initiating a new iteration of the BPM lifecycle.

As the Business-Process-Lifecycle forms the foundation for various process optimisation strategies and methodologies. Methods used to analyse, influence, or enable processes are inherently connected to one or more of these lifecycle steps. Notably, these methods often fall under the umbrella of PM, a field to be explored in the subsequent section.

1.2 Process Mining

PM, as elucidated by van der Aalst in [13], stands at the intersection of data science and process science. It serves as a comprehensive approach to glean insights and knowledge from event logs generated by operational processes within an organisation. While van der Aalst primarily highlights two subcategories, namely, discovery and conformance, a more comprehensive, and also popular classification involves three distinct categories: discovery, enhancement, and conformance, as evident in the literature review of P. Zerbino et al. [14]. This tripartite division provides a broader perspective on the diverse objectives and applications of PM, which we will therefore use as well. The relationship is also depicted in Figure 1.2.

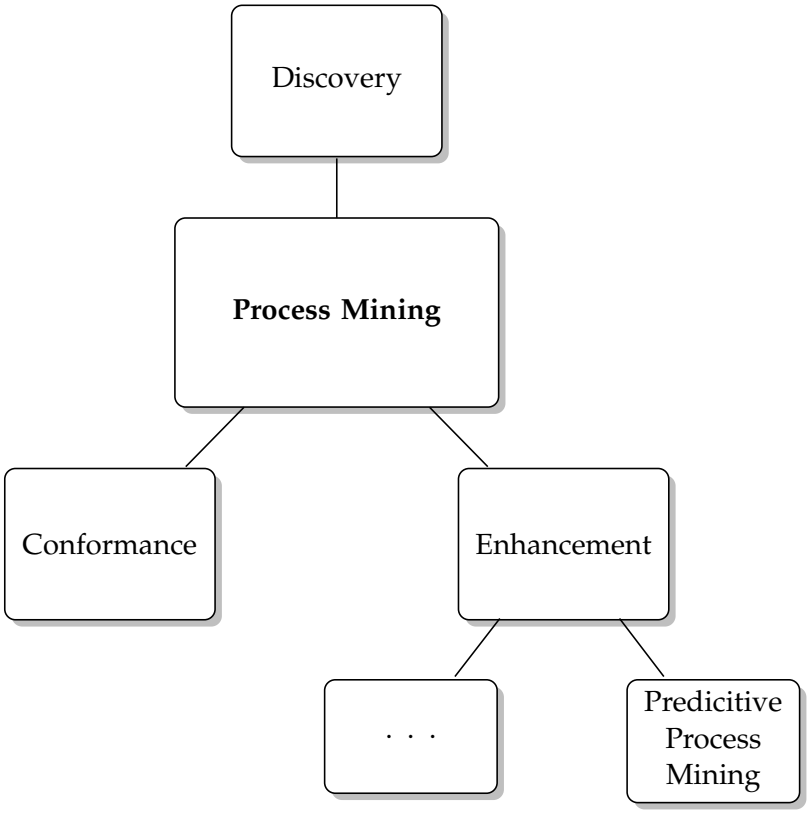


Figure 1.2: Categories in Process Mining

- **Discovery** pertains to the extraction of knowledge from event data - i.e. event logs - to create a process model, unveiling the implicit structures and patterns embedded in the data. This model illustrates the actual workflow of processes within an organisation based on historical data. Techniques such as Directly-Follows Graphs or the Alpha Algorithm are employed to automatically construct process models based on observed behaviour [15].

- **Conformance** analysis aims to assess the degree to which observed processes conform to predefined models or expectations, by evaluating the observed processes against them. Common approaches to estimate the Precision (i.e. how much of the recorded behaviour is captured by the model - does the model describe the data?) or the Fittness (i.e. how much of the modelled behaviour is recorded in an event log - does the data fit the model?) are Alignments or Token Replay, which highlighting deviations and potential areas for improvement [16]. As the mentioned comparisson with an event log suggests, Conformance involves comparing actual process executions with the anticipated paths, identifying deviations, and if applicable analysing the root causes of discrepancies.

- **Enhancement** in PM focuses on augmenting existing process models or creating more sophisticated models in order to improve process execution. This involves leveraging insights derived from event data to enhance the efficiency, effectiveness, or other performance metrics of a given process. Enhancement contains various subfields, including

PPM. In the context of PPM, the goal is to enhance models for usage during execution, forecasting future states of a process based on historical data, and optimising the decision-making process within ongoing operations. While PPM could theoretically fall under conformance when used to validate whether a model adheres to desired behaviour, the primary objective here is proactive improvement rather than reactive validation. PPM hich will be address further in Section 2.2

The division of PM into these three categories (Figure 1.2) provides a structured lens through which the diverse applications and methodologies can be comprehended, while also being relatable to the Life-Cycle steps mentioned in Section 1.1. Noticeably, each of these categories can be further subdivided, thereby demonstrating the intricate nature of PM as a discipline grounded in definition and methodology.

The field of enhancement, despite being the smallest within PM, as depicted in Figure 1.3,

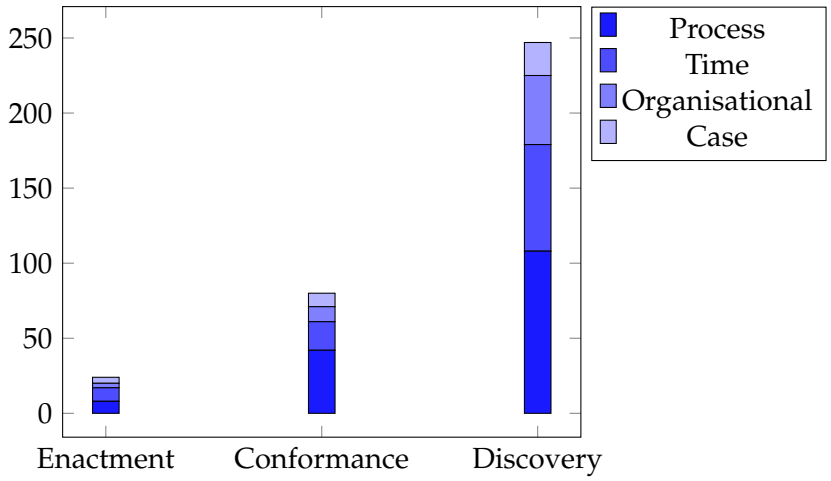


Figure 1.3: Number of PM-Papers (Categorical; including multiple peculiarities per paper), from [14]

based on the review conducted by P. Zerbino et al. [14], encompasses numerous subfields, including PPM. Despite its size, the field has garnered attention for its potential to proactively optimise processes. PPM, in particular, could emerge as an important field for researchers seeking to leverage historical data to anticipate and enhance the trajectory of business processes. This niche, while comparatively less explored, holds significant promise for organisations striving to stay ahead in the ever-evolving landscape of process optimisation.

The primary focus of this thesis lies within the Enhancement category of PM, specifically in the domain of PPM. In this research, the objective is to establish a base for enhancing a model for PPM, enabling it to make accurate and informed predictions about the continuation of a given process, by forecasting future states at any given point during execution.

1.3 Motivation

Recent advancements in NLP and Artificial Intelligence (AI) have propelled technology to unprecedented heights. To give just a few noteworthy examples: OpenAI's ChatGPT [1] , a sophisticated Large Language Model (LLM), and the integration of GPT-4 by Bing; or Google's Bard [2] which uses PaLM 2 by Google AI. The increasing number of AI-powered tools coupled with their growing availability for a greater extent of people underscores the transformative impact of these technologies on various domains [17, 18]. To comprehend the motivation behind the thesis, it's imperative to understand the significance of NLP, its intersection with AI, and its possible synergies with PPM.

NLP, an integral aspect of AI, aims to enable machines to understand, interpret, and generate human language. In order for any model or application to utilise, manipulate, or generate text in the form of natural language, it is indispensable to establish a connection between the text and the task that is intended to be accomplished on or with such input. To put it differently, NLP functions as an Application Programming Interface (API) to the task solving instance, rendering it a crucial component in the overall picture. This field encompasses a wide range of tasks, ranging from basic ones like text classification [19] and sentiment analysis [20] to more complex ones like language translation [21] and question-answering systems [22]. At its core, NLP aims to bridge the communication gap between humans and machines, thereby facilitating seamless interaction and comprehension. Consequently, enhancing models capabilities to generate text that resembles human language, answer inquiries, and engage in coherent conversations reflects a significant shift in language comprehension and generation capabilities. In its most basic form, NLP signifies the transition from human-readable text to machine-readable text and vice versa.

The motivation of the thesis is based on the realisation that the principles and techniques applied to NLP, particularly tokenisation, can be harnessed to enhance the abstraction and understanding of events within PPM. In the NLP domain, text is often encoded into sequences of tokens, forming the basis for various language-related tasks. Tokenisation, a fundamental process in NLP, involves breaking down text into smaller units, typically words or subwords, and ultimately projection them into a form that is usable within the program of the application (e.g. a number or a vector of numbers).

We will proceed to explain tokenisation by utilising a example:

$$T := \text{"Ein Vogel saß auf einem Ast, es regnete, er wurde nass.}$$
$$\text{Da kam der liebe Sonnenschein, es müssen zweiunddreißig sein."}$$

Definition 1. *A **Word** is defined as a sequence of one or more alphanumeric characters (letters or digits).*

$$W := \texttt{[a-zA-Z0-9]+}$$

Definition 2. *A **Sentence** S is defined as a sequence of one or more words, seperated by an Element of κ:*

[1] https://chat.openai.com/
[2] https://bard.google.com/chat

- *An optional seperator and an obligatory whitespace ("\n")*
- *Or bridging symbol ("-")*

and followed by an ending symbol (".", "!", "?") ω.

$$\kappa := (\,(\,\lambda\,|\,,\,|\,;\,|\,:\,|\,')\,\backslash n\,)\,|\,- \qquad , \lambda \text{ being the empty symbol}$$
$$\omega := !\,|\,?\,|\,.$$
$$S := \{\,(\,w\,z\,)^*\,w\,e\,|\,w \in W \wedge z \in \kappa \wedge e \in \omega\,\}$$

With Definition 1 and Definition 2 T, i.e. the aforementioned sentences, is representable as follows:

$$T = [s_1, s_2] \qquad , S_i \in S \wedge i \in \mathbb{N}$$

Consequently, making T abstractable into a sequence of sequences of words, removing the seperators and ending symbol.

$$T = [\langle w_1, w_2, ...\rangle \qquad , w_x \in W \wedge x \in \mathbb{N}$$
$$\langle w_i, w_k, ...\rangle] \qquad , i \wedge k \in \mathbb{N}$$

A simple tokenisation of T could be to map each individual existing word to a integer.

Word	**Associated Number**
Ein	1
Vogel	2
saß	3
auf	4
einem	5
Ast	6
es	7
regnete	8
er	9
wurde	10
nass	11
Da	12
kam	13
der	14
liebe	15
Sonnenschein	16
es	7
müssen	17
zweiunddreißig	18
sein	19

Table 1.1: Mapping the sequence of words to a sequence of integers

With the translation map depicted in Table 1.1, T can be tokenised to such an integer sequence.

$$\begin{aligned} T = &[\langle 1,2,3,4,5,6,7,8,9,10,11\rangle \\ &\langle 12,13,14,15,16,7,17,18,19\rangle] \end{aligned}$$

This abstraction is analogous to the manner in which event logs can be represented in PPM.

Definition 3. *A **Trace** T_r is defined as a sequence of one or more events $e_i \in E$ (set of all events) $\wedge\, i \in \mathbb{N}$:*

$$T_r := \langle\, e_i\,,\, e_k\,,\, \dots \rangle \qquad\qquad ,i\,,k\,,\, m \in \mathbb{N}$$

Definition 4. *A **Log** L is defined as a collection of one or more traces.*

$$L := [\, t_i^m\,,\, t_k^n\,,\, \dots] \qquad\qquad ,i\,,k\,,\, m\,,\, n \in \mathbb{N}$$
$$,m\,,n \text{ number of occurrences}$$

Using traces to abstract the data generated by a process during execution (i.e. event log) is a common approach, also used in other fields of PM, such as Alignments. A trace, a sequence of events, serves as an abstraction of the fundamental process and can be once again mapped to one or more tokens. Following the approach of before, we could again represent this sequence by mapping the events to a corresponding integer:

$$\begin{aligned} L_1 = &[\langle e_1, e_2, e_3\rangle^k && ,k,l \in \mathbb{N} \text{ (number of occurrences)} \\ &\langle e_1, e_3\rangle^l] && e_i \in \text{Set of all events in } L \wedge i \in \mathbb{N} \end{aligned}$$

$$\Downarrow$$

$$\begin{aligned} L_1 = &[\langle 1,2,3\rangle^k && ,k,l \in \mathbb{N} \text{ (number of occurrences)} \\ &\langle 1,3\rangle^l] \end{aligned}$$

The resemblance between language processing and PM serves as a catalyst for the exploration of DL and NLP techniques aimed at enhancing predictive capabilities within the domain of PM. The prospect of applying the principles of tokenisation and sequence analysis to event logs opens up avenues for PPM that leverage the rich insights embedded in historical process data. The thesis aims to take advantage of the conceptual and methodological progression of NLP related tasks, and apply language understanding techniques and a predictive model, contributing to the evolution of NLP applicable methods in the context of PPM.

2 Related Work

Towards the end of the 20^{th} century, the escalating enthusiasm in diverse domains such as Data Mining and Process Modelling, among others, resulted in the increasing popularity of Petri-Nets, a mathematical notation that is utilised to depict and capture the intricate details of processes and their dynamic. Within this setting Agrawal et al. [23] were one of the first to propose the fusion of data mining and process modelling, publishing their research report titled "Mining Process Models from Workflow Logs" in 1997, and, one year later, their identically named paper. The term "Process Mining" itself was introduced by Wil van der Aalst, who has been a prominent figure in the evolution of this discipline. To my best knowledge, the initial use of the term can be traced back to his publication titled "Process design by discovery: Harvesting workflow knowledge from ad hoc executions" [24]. Since then, PM has developed into a scientific area, focusing on the abstraction of knowledge from event data, particularly execution data.
As the field matured, a diverse range of perspectives were incorporated, extending beyond the mere workflow to encompass elements such as time, costs, and case considerations [12], consistently aiming to employ data mining techniques to extract a diverse range of information, enhancing the comprehension of processes. The proliferation of research and practise in PM has led to the development of a plethora of tools, catering to various needs within the discipline. Notable examples of tools that emerged over the years are ranging from open-source options like ProM [25] and libraries such as PM4Py [1] to commercial solutions like Disco [2].
In 2006, the IEEE established a non-commercial association, the "Task Force on Process Mining," which subsequently published the Process Mining Manifesto [26], a foundational document providing guidelines and addressing challenges for individuals involved in PM tasks. Within the manifesto, PM is defined as

> Process mining: techniques, tools, and methods to discover, monitor and improve real processes (i.e., not assumed processes) by extracting knowledge from event logs commonly available in today's (information) systems [26, p. 15].

Additionally, prediction was recognised as part of a challenge in PM, specifically under the category "Providing Operational Support" [26]. Further contributing to standardisation efforts, in 2016, the IEEE introduced the XES eXtensible Event Stream (XES) standard [27]. This XML-based schema aims to provide a unified format for storing data produced during process execution. The introduction of this standard facilitated interoperability between various tools and systems, promoting a more cohesive approach to handling event data in the context of PM.
In summary, the historical trajectory of PM reflects a dynamic evolution from its early integra-

[1] https://pm4py.fit.fraunhofer.de/
[2] https://fluxicon.com/disco/

tion of data mining and process modeling principles to its current status as a thriving scientific discipline. Along with the alleged doubling of the PM market every 18 months, as stated in [12], the growing significance and acceptance of PM techniques across various domains demonstrates a persistent and expanding interest from industry in the field.
Given the vast expanse of PM, we will proceed this chapter by outlining my research approach employed to identify relevant literature. Following this, we delve into different approaches in the broader PPM domain, subsequently narrowing the focus to approaches within the sub-field of PPM that leverage DL techniques. Finally, the application of transformers, a specific class of DL models, in PPM will be explored.

2.1 Research approach

The research methodology that we employed to identify material of interest involved a comprehensive examination of literature and resources in the domains of PM, PPM, and DL. Leveraging search engines such as Google Scholar [3] and Semantic Scholar [4] , we utilised a set of key terms including "Process Mining," "Predictive Process Mining," "Process Monitoring," "Transformer," "Sequence Prediction," "Prediction in Deep Learning," and "Deep Learning." We initiated the search by referring to the seminal work in PM, such as those consolidated by van der Aalst [12]. By employing a "snowball" approach, we systematically examined the papers that cited these initial works, as well as delved into their references, thereby broadening the scope of pertinent literature. This iterative procedure facilitated the identification of significant contributions and established a robust foundation for the thesis. Additionally, we extended the search beyond traditional academic databases by exploring open-source repositories on platforms like GitHub [5]. The examination of projects pertaining to DL models, such as GPT-2 [28, 29] and nanoGPT [30], has provided valuable insights into the practical implementations of sequence prediction and predictive modelling. To gain a deeper understanding of the technical aspects, we delved into the documentation and instructional material of widely used DL frameworks such as TensorFlow [31] and PyTorch [32]. This approach not only enriched the theoretical understanding of DL but also provided practical insights into the implementation details of models like Transformers. The combination of academic literature review, citation analysis, exploration of open-source repositories, and hands-on engagement with DL frameworks collectively shaped a well-rounded research approach. This methodology ensures a comprehensive exploration of the existing knowledge landscape, fostering a nuanced understanding of the interplay between PM, PPM, and contemporary DL techniques, particularly Transformers.

2.2 Predictive Process Mining

PPM stands as an integral component of PM, specifically within the broader scope of monitoring, as highlighted in Section 1.2. Despite the term being sometimes interchangeably referred to as Predictive Process Monitoring, the distinction is subtle, and both convey the central idea of anticipating and adapting to changes in a process during its execution. The term "Predic-

[3] https://scholar.google.com/
[4] https://www.semanticscholar.org/
[5] https://github.com/

tive Process Mining" is entrenched in the literature, underscored by its usage in seminal works such as [13]. Hence, we will proceed to use the term PPM. In essence, PPM involves mining predictions from event logs, enabling organisations to enhance processes in a forward-looking manner [13]. This prospective approach, in contrast to retroactive methodologies such as Conformance Checking or Process Discovery (i.e. backwards-looking [13], Section 1.1), positions PPM as a catalyst for adapting processes during runtime. The overall goal is to empower organisations to leverage predictions of state changes in real-time, facilitating dynamic adjustments and optimisations [33], making it a powerful tool for organisations seeking real-time insights.

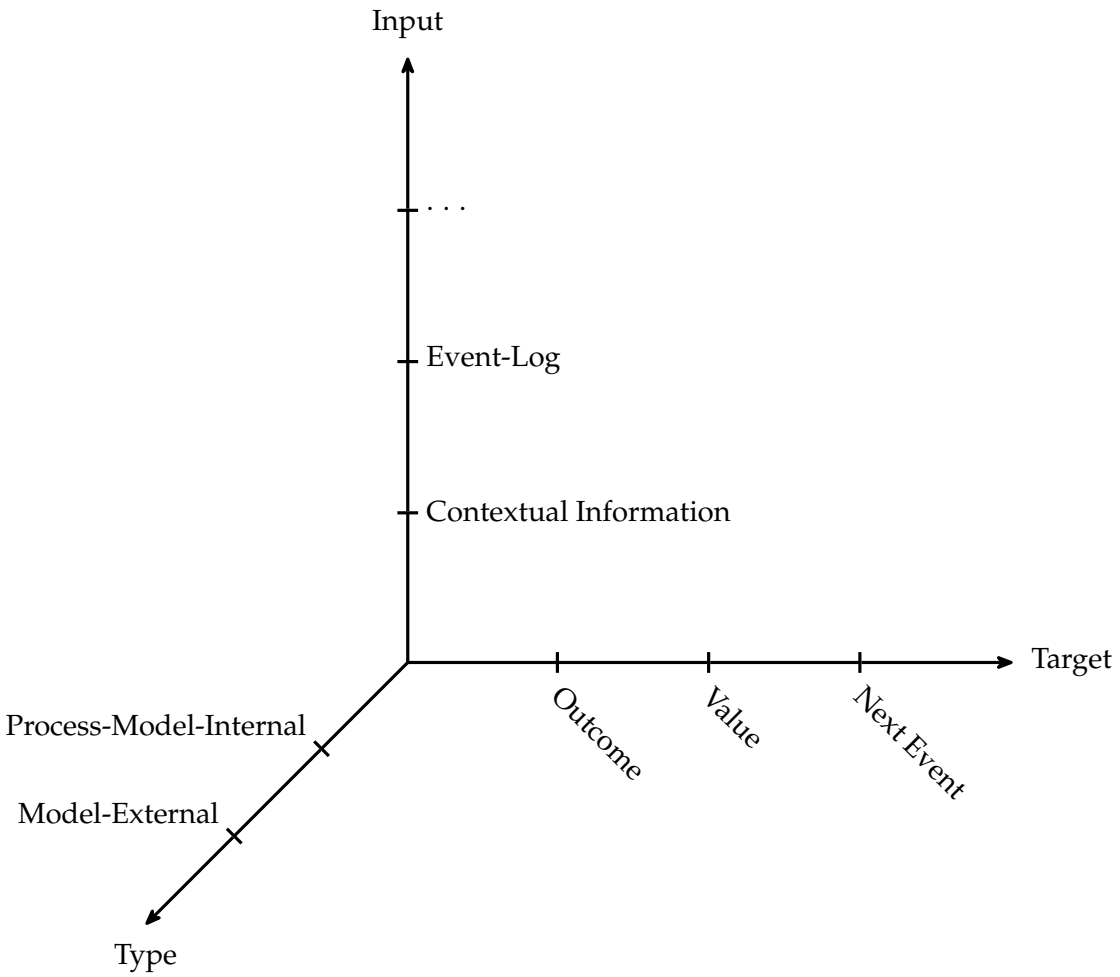


Figure 2.1: Dimensions of PPM, based on the categories mentioned in [33, 34]

PPM approaches can be categorised along three dimensions, providing a systematic framework for understanding the diverse methodologies within this field. These dimensions include the input used for prediction, the target of prediction, and the type of approach employed. Figure 2.1 visually illustrates these dimensions.

- The **Input-Dimension** refers to what is used for the prediction. That could essentially be any combination of data relevant to the process. To give a few examples:
  Traditional PPM approaches utilise historical event data or event logs to make predictions about future states of a process. The richness of event data allows for a comprehensive understanding of past process executions, forming the basis for predictive modeling.
  Some PPM approaches augment event data with additional contextual information, enhancing the predictive power of models. Contextual factors, such as external environmental conditions or organisational context, can significantly impact the evolution of a process. [35]

- The **Target-Dimension** pertains to what is being predicted. It could be the next event in the sequence - a fundamental aspect of PPM, aiming to anticipate the subsequent steps in the process. Or it could be the outcome of the process (i.e., the result) - especially relevant for processes where the ultimate goal is a specific outcome rather than individual events. Furthermore, any value related to the events or the entire process, such as cost or time, effectively broadening the scope of prediction beyond the sequence of events to encompass holistic process characteristics. For clarity, we will refer to values related to events as "event properties" and those related to the entire process as "process properties."

- The **Type-Dimension** categorises approaches into two main types: Model-Internal and Model-External. Model-Internal approaches try to enhance existing models to include predictive elements, effectively integrating prediction into the core structure of the model. Examples include utilising Transition Systems as a Control-Flow Model [36], employing Sequence Trees [37], or incorporating Stochastic Petri-Nets [38]. Model-External approaches employ separate statistical or machine learning models to make predictions. The prediction model collaborates with the original process (model) to dynamically predict state changes. Model-external approaches can be further categorised based on the nature of the prediction task, with some treating it as a classification problem [34], while others explore sequence generation for predictive purposes, a category that aligns with the approach of this thesis, which we will discuss in Section 2.2.1.

In summary, PPM, with its focus on forward-looking adaptations during runtime, has emerged as a transformative paradigm within PM. Within the realm of PPM, various methodologies and techniques have been explored. A considerable body of literature categorises these approaches based on their underlying mechanisms and goals. The dimensions of input, target, and type provide a structured lens for comprehending the diversity of PPM approaches. PPM, being an evolving field, encompasses both established and emerging methodologies. Machine learning techniques play a pivotal role in many PPM approaches. These approaches treat PPM as a supervised learning task, where historical event data serves as the training set, and the objective

is to train a model to predict a future state - in parts or as a whole.

2.2.1 Deep Learning in Predictive Process Mining

DL, as a subset of ML, has significantly impacted the landscape of PPM. When separating DL from conventional (shallow) Machine Learning techniques, it is noteworthy that the former employs multi-layer neural networks. The distinctive architecture of DL models confers the capability to handle extensive and high-dimensional data, automatically capturing intricate non-linear relationships, and representing data at multiple levels of abstraction [**?**, 39]. While exploring all aspects of DL is beyond the scope of this thesis, it is crucial to establish a foundational understanding of its principles, advancements, and historical evolution within the context of PPM.
For a comprehensive exploration of DL origins, [40] provides an in-depth resource. we will proceed to provide a overview of the historical advancement of DL, highlighting significant milestones and the evolution of neural network architectures. This serves as a foundation for understanding the various neural networks employed in PPM and their distinct methodologies.
The roots of DL can be traced back to the conceptualisation of artificial neurons by McCulloch and Pitts in 1943 [41], which laid the groundwork for neural networks. However, it was Hebb's introduction of the Hebbian Learning Rule in "The Organization of Behavior" [42] in 1949 that laid the groundwork for neural network theory, marking him as the "father of Neural Networks". The subsequent years witnessed pivotal advancements, including the introduction of the Perceptron by Rosenblatt in 1958 [43] - an early form of neural network architecture - and the development of backpropagation by Werbos in 1974 [44, 45], a pivotal algorithm that enables the training of neural networks by iteratively adjusting weights based on prediction errors. Before that, the term 'backpropagation' was not recognised with the gradient decent algorithm. The 1980s brought forth Neocogitron, a model by Fukushima that laid the foundation for Convolutional Neural Networks (CNN), revolutionising image processing [46]. Jordan defined Recurrent Neural Networks (RNN) in 1986, which excel in handling sequential data due to their ability to retain and utilise information from previous steps. In 1997, Hochreiter and Schmidhuber introduced Long-Short-Term-Memory Networks (LSTM) to address the vanishing/exploding gradient problem in RNNs [47]. LSTMs - a sub-type of RNNs - enable the retention of long-term dependencies in sequential data, by using so called LSTM-Cells. The year 2017 marked a significant milestone with the introduction of the Transformer architecture by Vaswani et al., a model that embraces attention mechanisms [21]. Transformers have since become a fundamental component in numerous applications, particularly in the realm of NLP.

DL, with its ability to automatically learn hierarchical representations from data, has become a transformative force in PPM. Figure 2.2 illustrates the various types of neural networks relevant to PPM.

- The simplest form of neural network, **FeedForward Neural Networks** (FFNN), consist of an input layer, one or more hidden layers, and an output layer. These networks commonly excel in tasks such as classification, pattern recognition, or regression [48]. However, their limitation lies in the absence of memory, hindering their effectiveness in handling sequential data [48]. Problems such as vanishing or exploding gradients further im-

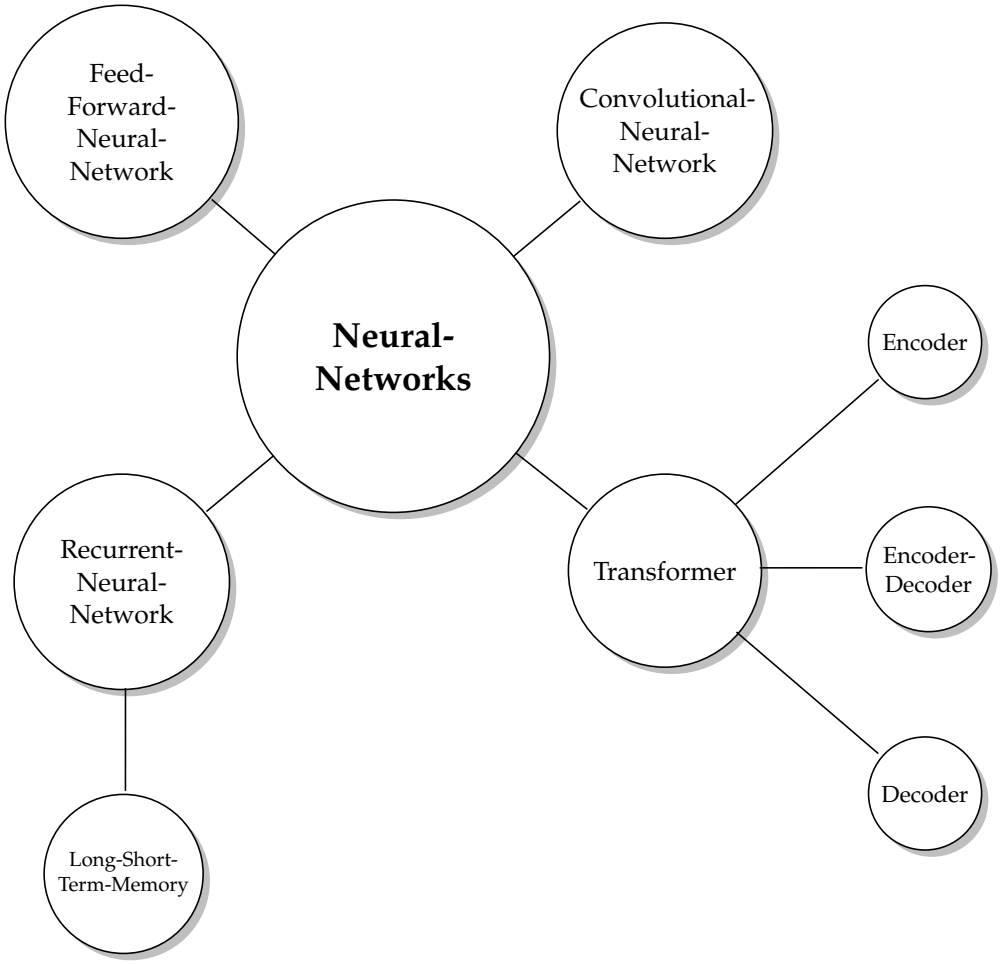


Figure 2.2: Types of Neural-Networks in the context of PPM

pede the training of deep FFNNs [48]. Nonetheless, FeedForward networks have paved the way for more sophisticated architectures [48].
Despite these limitations, FFNNs have found application in PPM, as evidenced by approaches such as [49] and [50]. These approaches leverage FFNNs for capturing complex patterns in event data.

- Designed for grid-like data such as images, **Convolutional Neural Networks** (CNN) employ convolutional layers to learn spatial hierarchies of features adaptively. They are highly effective in image analysis, object detection, and recognition tasks. The utilisation of parameter sharing in CNNs facilitates the detection of patterns irrespective of their location in the input space. Nevertheless, challenges may arise with large-scale data, requiring significant computational resources. [51, 52]
  In the context of PPM, approaches like [53] and [54] leverage CNN architectures, harnissing the pattern recognition capabilities of CNNs to make predictions based on event sequences.

- **Recurrent Neural Networks** (RNNs) are specifically designed to process sequential data by preserving a hidden state that captures information about previous inputs. Despite this, they are afflicted by the vanishing gradient issue, which hinders their capacity to effectively model long-range dependencies in sequential data. **Long-Short-Term-Memory** (LSTM) were introduced as a solution to this issue, incorporating memory cells with gating mechanisms. These mechanisms enable the network to selectively retain or forget information, thus addressing the issue of vanishing gradients. [55]
  In PPM, approaches such as [56] and [57] leverage LSTM networks. These approaches utilise the memory capabilities of LSTMs to capture and model temporal dependencies within event sequences.

- The **Transformer** architecture represents a groundbreaking paradigm shift in neural network design. Originally designed for NLP tasks like machine translation [21], Transformers move away from sequential processing to self-attention mechanisms. Attention mechanisms enable the model to weigh input sequence elements differently during processing, allowing it to focus on relevant information dynamically. This simultaneous processing of the entire input sequence facilitates improved parallelisation and training efficiency [21, 58].
  Transformers have become the foundation for state-of-the-art models in various domains, showcasing their versatility and effectiveness. Their success extends to Computer Vision [59], Speech Processing [60], and, as importantly, PPM. The Transformer's capacity to capture dependencies across the entire input sequence aligns seamlessly with the dynamic and sequential nature of PM. Furthermore, the Transformer architecture, the focal point of this thesis, will be explored in greater detail in Section 2.2.1.1.

This historical and conceptual overview provides a glimpse into the evolution of DL and its various Neural Network architectures. The subsequent sections of this thesis delve into the utilisation of transformers in sequence prediction within the context of PPM. With the approaches mentioned so far, as well as the approaches that will be mentioned in the next sections, it becomes evident that the evolution of DL has left an indelible mark on PPM methodologies, as most DL approach are leveraged by different PPM approaches, a trend that probably will continue.

In conclusion, the evolution of neural network architectures in PPM reflects a dynamic interplay between the specific demands of the domain and the capabilities of these models. While traditional architectures like FFNNs, CNNs, and LSTMs have paved the way for understanding sequential dependencies, the Transformer architecture represents a paradigm shift that promises to revolutionise the predictive modelling landscape in PM.

2.2.1.1 Transformer and Predictive Process Mining

As with most concepts, there are different Transformer architectural types. This Section seeks to provide a nuanced understanding of the existing Transformer architectures and their potential applications within PPM, therefore laying the groundwork for the subsequent exploration of a

Decoder-Only model in this thesis. As depicted in Figure 2.3, Transformers can be categorised into three main types: Encoder-Only Models, Decoder-Only Models, and Encoder-Decoder Models. Each type exhibits distinctive characteristics, catering to specific use-cases (adapted after [61, 62]):

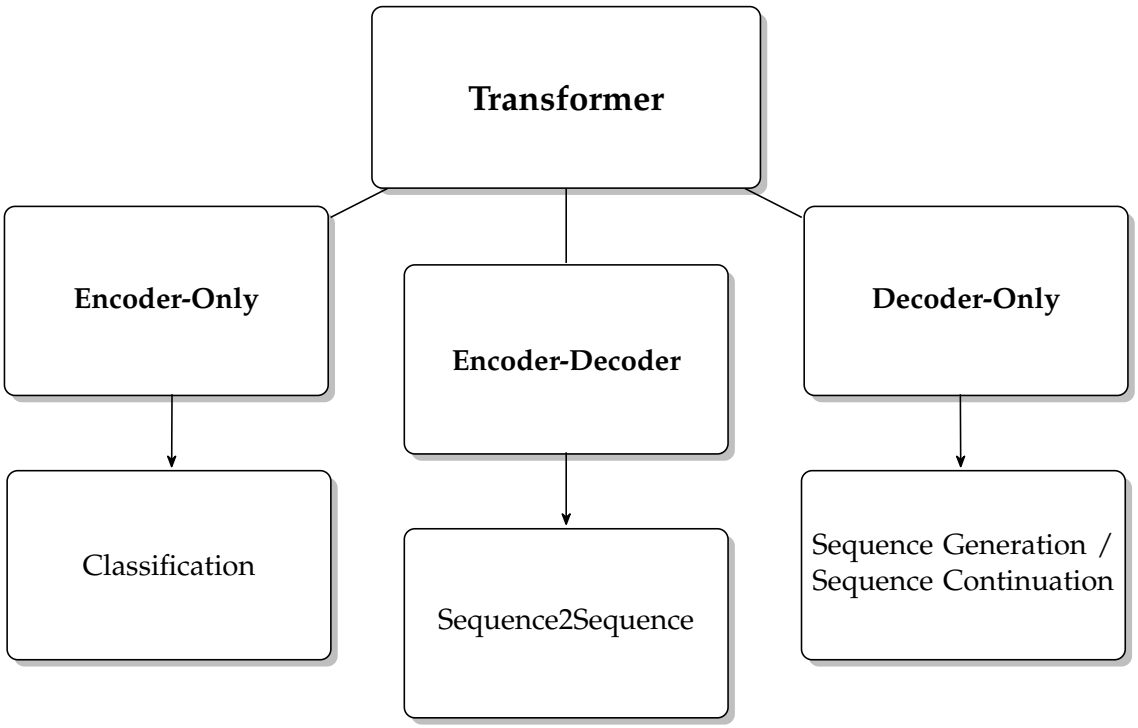


Figure 2.3: Transformer Architecture Types

- **Encoder-Only** models take a sequence as input and project it into a fixed-length vector, essentially creating an embedding. While these models comprehend the input sequence well, their ability to generate it is limited. They are particularly suited for classification tasks. Bidirectional Encoder Representations from Transformers (BERT) [63] stands as a prominent example of an Encoder-Only model. BERT's bidirectional approach enables it to capture context and relationships within a sequence, making it effective for various natural language understanding tasks.

- **Decoder-Only** models generate one output token at a time based on the context provided by the input. These models stand out in tasks where prompt-based generation is essential, as they can continue or complete the input sequence. ChatGPT [22] exemplifies a Decoder-Only model. Trained through a generative language modelling approach, ChatGPT is adept at producing coherent and contextually relevant responses in natural language conversations.

- The original Transformer architecture [21] employs both encoder and decoder components, making it a versatile sequence-to-sequence model. **Encoder-Decoder** models are well-suited for tasks like translation or summarisation, where understanding the input

sequence and generating a corresponding output sequence are integral. Bidirectional and Auto-Regressive Transformers (BART) [64] represents an Encoder-Decoder model. BART's bidirectional design allows it to efficiently generate coherent outputs by considering both past and future context in the input sequence.

It is important to note that while these categorisations offer a general rule of thumb, there are no strict limitations on using different Transformer architectures for diverse tasks. The adaptability of Transformers has led to their application in a wide array of domains beyond their initially proposed use-cases.
The utilisation of Transformer models in PPM marks a significant stride in the exploration of advanced DL architectures for sequence prediction within the context of business processes. Several pioneering approaches have employed Transformer models in PPM, showcasing the versatility and effectiveness of this architecture.
Z. A. Bukhsh et al. were the first to propose the usage of a Transformer in 2021, introducing the concept of ProcessTransformer [65]. Their proposal is one of the pioneering works in employing Transformer models for predictive tasks in processes. The ProcessTransformer approach adopts a decoder-only architecture with an embedding layer, showcasing its capacity to predict the next activity, next event time, and remaining time. The ProcessTransformer's decoder is adept at learning intricate patterns within event sequences, offering a versatile framework for predictive analytics in dynamic processes.
G. Rivera Lazo et al. proposed a Multi-attribute Transformer, introducing an encoder-decoder Transformer architecture for predictive process analytics [66]. The encoder of this model transforms incomplete multi-attribute cases into trace abstractions, while the decoder generates a probability distribution for subsequent predictions. By combining the encoder's ability to map complex cases into trace abstractions with the decoder's predictive capabilities, this approach showcases the potential of Transformer models in handling multi-dimensional (i.e. attributes) data for accurate predictions.
Ni et al. introduced a Hierarchical Transformer Model designed to capture contextual information in a hierarchical manner [67]. This innovative architecture comprises four distinct layers, with the initial three layers functioning as transformers, each focusing on different aspects of the process. The first layer encodes events and their properties, the second layer employs cross-self-attention to capture context within sub-sequences, and the third layer utilises self-attention with positional encoding to reflect case context. The final layer transforms the encoded information into predictions for state changes, highlighting the effectiveness of hierarchical attention mechanisms in modeling complex process dynamics.
Wang et al. proposed a Multi-View Transformer that addresses the multi-perspective nature of process data [68]. This approach involves three phases: Data Preparation, View Extraction, and Prediction. In the data preparation phase, the event log is processed to extract information for three distinct views: Activity, Time, and Resource. Each view is separately embedded, encoded, normalised, and concatenated before serving as input to the Transformer's decoder. This multi-view approach allows the model to simultaneously consider different perspectives, enhancing its ability to make comprehensive predictions.

While these approaches share the foundational concepts of DL and Transformer architectures, their unique designs underscore the adaptability of Transformer models to diverse PPM scenarios and perspectives. The success of each approach hinges on its ability to effectively capture

temporal dependencies, context, and multi-dimensional information inherent in business processes.
As the field of PPM continues to evolve, future research directions may explore hybrid architectures that combine Transformer models with other DL approaches. Additionally, efforts to enhance interpretability and explainability of predictions could contribute to the broader adoption of these advanced models in real-world business settings.
In Section (3.2), the focus will shift to the specific architecture chosen for this thesis, providing a detailed exploration of its design principles and how it aligns with the overarching objectives of enhancing predictive capabilities in business processes.

3 Concept and Design

This section provides a detailed exploration of the conceptual and design considerations underlying the proposed transformer-based architecture for PPM. The integration of proven Transformer principles with customised adaptations is intended to improve the model's performance in capturing and predicting sequences within complex process data. The objective is to build upon established Transformer architectures while introducing custom adaptations to suit the intricacies of predictive tasks in PM.

To ensure a robust and proven foundation, the architecture will draw inspiration from existing successful Transformer models. The selection of a base architecture holds great significance, and a range of factors that should be taken into account include model complexity, efficiency of the attention mechanism, and suitability for the particular use case. The architecture will preserve the fundamental structure of Transformers.

Rather than extensively modifying the architecture, emphasis will be placed on adapting the format of the input through custom tokenisation. The rationale behind this choice is outlined in Section 3.2.2.1. Implementing a custom tokenisation will enable the model to effectively capture the subtleties of process data, thereby facilitating enhanced learning and prediction.

A pivotal component of the Transformer architecture is the attention mechanism, enabling the model to weigh the significance of different parts of the input sequence when making predictions. The attention mechanism enhances the model's ability to capture long-range dependencies and contextual information. In the context of PPM, understanding how the attention mechanism operates is essential for interpreting the model's predictions and ensuring meaningful insights are derived from the process data. We will dive more into Attention in Section 3.2.3.1. The architectural description will be accompanied by figures that will visually represent the components of the proposed architecture. These figures aim to enhance comprehension and provide a visual guide to the structure and flow of information within the model.

3.1 Current Architectures

Since the groundbreaking introduction of Transformers by Vaswani et al. [21], numerous Large Language Models (LLM) have been introduced, transforming the landscape of NLP. Notable models include open-source initiatives like BERT [63] and Llama 2 [69], as well as proprietary models such as GPT-4 [70], GPT-3 [71], and Palm [72]. A significant trend in recent LLM developments is the rise of open source models, providing accessible frameworks for researchers and practitioners.

This surge in open source availability, as highlighted in [73], empowers users to leverage pre-trained models as foundational blocks for various NLP tasks. The idea of using pre-trained models that may not be task-specific but are fine-tuneable has gained prominence [74, 75]. This approach eliminates the need for designing task-specific architectures, making it more versatile and applicable across different domains. In essence, the exploration of current LLM architectures provides a foundation for the proposed approach in Process Science. The adaptation and fine-tuning strategy aim to harness the advancements in pre-trained models and apply them to the specialised domain of PPM.

However, the diverse array of available pre-trained models is primarily tailored for NLP tasks, and their direct applicability to the domain of Process Science is limited. To bridge this gap, we propose an approach that involves adapting a general transformer-based architecture, inspired by open source models such as GPT-2 [29] or NanoGPT [30], that is then adapted to align the model with the intricacies of process data and the specific requirements of PPM. The suggested strategy involves pre-training the adapted model on diverse datasets to imbue it with a broad understanding of the 'Process-Language' and context. While pre-training provides a solid foundation, the subsequent fine-tuning process narrows the model's focus to the intricacies of a certain and fixed process event log, specific to a use-case or organisation This fine-tuning phase enables the model to learn from domain-specific data, tailoring its predictions to the nuances of process sequences, directly derived from its target. Given the constraints of a bachelor thesis, the overall goal of deploying a universally applicable, pre-trained, and fine-tuned model for companies is acknowledgeable but deemed beyond its scope and my capabilities. Instead, the focus will be to showcase the proposed methodology on a smaller scale, providing valuable insights into the adaptability of a general transformer-based architecture within the context of PPM. This will be accomplished by initially pre-training a GPT-Based model on a collection of event-logs, followed by adjusting and fine-tuning it for a specific process.

3.1.1 Generative Pre-Trained Transformers

The concept of Generative Pre-Trained Transformers (GPTs) was introduced by Radford et al. in 2018, marking a significant advancement in NLP models, particularly within the Transformer architecture [76]. OpenAI, the organisation behind this innovation, leveraged a decoder-only Transformer, essentially a Large Language Model (LLM), pre-trained on diverse datasets such as BooksCorpus [77], and later on fine-tuned and trained it again for the use on specific tasks. While GPTs may vary in their details, they all embody a set of characteristics: GPT models follow an autoregressive approach, predicting the next token in a sequence based on the preceding tokens. This characteristic aligns with the nature of sequential data, allowing the model to generate coherent and contextually relevant sequences. The generative capability of GPT models denotes their proficiency in creating meaningful and contextually appropriate sequences, primarily text. This feature makes them valuable in various natural language understanding tasks. During pre-training, GPT models process the input sequence in a unidirectional manner, specifically from left to right. This autoregressive training approach contributes to the model's ability to understand and generate sequences based on the context provided by preceding tokens. GPT models utilise a masking mechanism, where each token is predicted without any token masking. The model generates tokens one by one based on the context established by preceding tokens. The use of a causal or autoregressive mask ensures that, during training, the

model only attends to tokens that precede the current position, distinguishing it from bidirectional models.

While the fundamental concept of GPTs has remained consistent, subsequent iterations by OpenAI, such as GPT-3, introduced advancements achieved with techniques such as zero-shot or few-shot learning [71]. These improvements expanded the model's capabilities, allowing it to perform tasks with minimal examples or even in a zero-shot manner. An approach of letting a model learn the underlying concepts from pre-training on a large amount of data and using this knowledge to continue an input sequence closely aligns with the task addressed in this thesis. As GPTs have proven effective in understanding and generating sequential data, their underlying principles provide valuable insights for designing and implementing the proposed predictive process mining model.

3.2 Proposed Architecture

The proposed architecture, illustrated in Figure 3.1, comprises a streamlined data pipeline encompassing two fundamental steps: preprocessing and the transformer. Within the transformer, various components collaborate to execute the predictive process mining task effectively. The primary components include the decoder block, tokeniser, and embedding / encoding layer. The architecture begins with a data pipeline, acknowledging the importance of quality data preprocessing in enhancing model performance. This pipeline incorporates essential preprocessing steps to prepare the raw process event logs for the subsequent transformer model. The architectural decision to adopt a decoder-only model is deliberate, driven by the specific requirements of the predictive task. The goal is to predict the next event in a sequence, making a decoder-only model apt for this forward-looking task of continuation of process event sequences. This model type has demonstrated effectiveness in sequence continuation, aligning seamlessly with the predictive process mining objective. The subsequent sections will delve into more intricate details, thoroughly addressing each component.

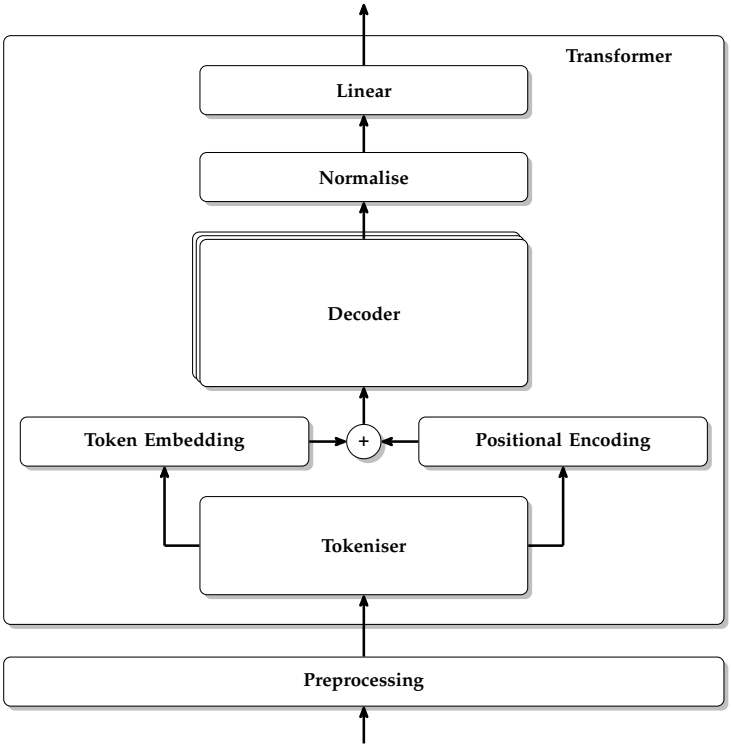
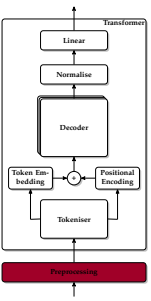


Figure 3.1: Proposed Architecture



3.2.1 Preprocessing

A common form of event logs is the XML-based XES-Standard [27]. In order to ensure the universal compatibility of these logs with the model, it is imperative to perform a preprocessing step. The primary objective is to convert the diverse information encoded in the event logs into a standardised format, allowing the model to seamlessly handle event logs from various processes. The XES-Standard employs XML tags such as <global>, <classifier>, and attributes to represent information about events and traces. The preprocessing step involves extracting and abstracting relevant details from these tags, creating a uniform representation for each event log. The desired output is a list of triples for each trace within a log, each containing these three elements: the identifier (e.g., the name of the event), a list of event properties, and another list of trace properties.

To achieve this, the following steps are undertaken:

1. **Event Classifier:** For each log, the event classifier is utilised to extract different peculiarities and create a lookup table, mapping them to unique integers. This ensures that event types are encoded consistently across various logs, allowing the model to understand and generalise across different processes.

2. **Global Attributes (Event Scope):** For each log, the global attributes with a scope-level of event are employed to extract event properties. These properties are then related to a numerical representation using a predefined translation table. Attributes defined in the XES-Standard, such as "concept:name," "lifecycle:transition," etc., are mapped to numerical values as outlined in Table 3.1.

3. **Global Attributes (Trace Scope):** Similarly, for each log, the global attributes with a scope-level of trace are used to extract trace properties. Each trace property is associated with every event within the trace. The values are translated according to the predefined mapping in Table 3.1.

The resulting lists of event and trace properties are then used as inputs for the tokeniser. To ensure a consistent length for these lists (currently set to 1000, subject to further optimisation), missing or undefined attributes are masked with the value 0. This preprocessing step is crucial for creating a standardised and informative input format for the model, enabling it to make context-aware predictions across diverse processes.

Attribute:	Translation rule:
String	If classifiable create a lookup table of all peculiarities and map each value to its corresponding number. Otherwise ignore.
Date	Convert to Unix Time.
Int	Preserve value.
Float	Preserve value.
Boolean	Map *FALSE* to -1 and *TRUE* to 1.
ID	Create a lookup table of all peculiarities and map each value to its corresponding number.
List	Unpack the list, to make it shallow, while preserving the order. Each element will be translated individually.
Container	Sort entries alphabetically (key), determining the order to unpack, to make shallow. Each element will be translated individually.
Nested Attributes	Unpack all child-elements, to make it shallow, while preserving the order. Each element will be translated individually.

Table 3.1: Maping for XES-Attributes to numerical values

To exemplify Table 3.1, let us look at an example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<log xes.version="2.0" xes.features ="arbitrary-depth"
    xmlns="http://www.xes-standard.org/">
    <extension name="Concept" prefix ="concept" uri ="http://www.xes-standard.org/
    concept.xesext" />
    <extension name="Time" prefix ="time" uri ="http://www.xes-standard.org/time.
    xesext" />
    <global scope="trace">
        <string key="concept:name" value="Trace" />
    </global>
    <global scope="event">
        <string key="concept:name" value="Event" />
        <date key="time:timestamp" value="1970-01-01T00:00:00.000+00:00" />
        <int key="event-prop" value="" />
    </global>
    <classifier name="Activity" keys="concept:name" />
    <trace>
        <string key="concept:name" value="First Trace" />
        <event>
            <string key="concept:name" value="A" />
            <date key="time:timestamp" value="1980-01-01T00:00:00.000+00:00" />
            <int key="event-prop" value="42" />
            <boolean key="not-global" value="false" />
        </event>
        <event>
            <string key="concept:name" value="B" />
            <date key="time:timestamp" value="1990-01-01T00:00:00.000+00:00" />
```

```
            <int key="event-prop" value="24" />
        </event>
    </trace>
    <trace>
        <string key="concept:name" value="Second Trace" />
        <event>
            <string key="concept:name" value="B" />
            <date key="time:timestamp" value="2000-01-01T00:00:00.000+00:00" />
            <int key="event-prop" value="4224" />
        </event>
    </trace>
</log>
```

Listing 3.1: Example XES Event-Log

Following the rule provided by Table 3.1, the Preprocessing yields the following result:

concept:name	
Trace	0
First Trace	1
Second Trace	2

Figure 3.2: Look-up Table for Traces

concept:name	
Event	0
A	1
B	2

Figure 3.3: Look-up Table for Events

concept:name	
Event	0
A	1
B	2

Figure 3.4: Look-up Table for Classifier

Figure 3.5: Produced Lookup-Tables in Preprocessing

$$\begin{aligned} \text{With log } L = &[[[1, [1, 315532800000, 42], [1]], && \text{, first event in first trace} \\ &[2, [2, 631152000000, 24], [1]]], && \text{, second event in first trace} \\ &[[2, [2, 946684800000, 4224], [2]]]] && \text{, first event in second trace} \end{aligned}$$

red $\doteq$ Event Classifier, blue $\doteq$ Event Properties, green $\doteq$ Trace Properties

3.2.2 Encoding

The choice of encoding method significantly influences how information is represented and processed within the model. One common approach to encoding categorical data is one-hot encoding, where each category is represented by a binary vector. While one-hot encoding is straightforward and easy to understand, it comes with certain drawbacks, especially in scenarios with numerous categories. Each category is represented by a binary vector with all zero values except for the index corresponding to the category, leading to sparse representations [78]. The dimensionality of these vectors grows with the number of categories, potentially resulting in computational inefficiencies and a lack of capturing relationships between categories.

Embedding, as proposed in the original Transformer paper [21], offers an alternative that addresses the limitations of one-hot encoding. In the embedding approach, categorical variables are mapped to continuous vectors of fixed size known as embeddings. These embeddings are not predefined, but are learned during the model training process. Embedding significantly reduces dimensionality compared to one-hot encoding. Instead of using a high-dimensional binary vector, embeddings represent categories in a continuous vector space of lower dimensionality, capturing semantic relationships and similarities between different categories [79]. This is crucial in PPM, where understanding the nuanced connections between various events and properties is essential for accurate predictions. Furthermore, embeddings are trainable parameters of the model, allowing the model to adapt and learn the most relevant representations for the given PPM task.

3.2.2.1 Tokeniser

Tokenisation, a fundamental step in NLP, involves breaking down sequences or subsequences into their smallest meaningful units, known as tokens [80, 81]. While traditional tokenisers like Google's SentencePiece [82] and OpenAI's tiktoken [83] are tailored for NLP tasks in natural language, the task at hand in PPM sets the stage for a custom tokenisation approach.

In the context of PPM, the sequences to be tokenised are the traces in a log, where each event essentially represents a "word." Just like natural language, where words can have different declination or conjugation, events in a process log can have various attributes, i.e. event properties and trace properties. These nuances need to be captured in the tokenisation process to allow the model to learn the relationships between events, their properties, and trace properties without explicit guidance. The model needs to be able to learn the differences between two different peculiarities of the same type of event, without losing the knowledge that it is nevertheless the same type of event.

To achieve this, a custom tokenisation strategy is proposed. Each event is translated into three distinct tokens, each representing a different aspect:

1. **Event Type Token:** Represents the type or category of the event. Encodes the fundamental nature of the activity captured by the event. Enables the model to understand the essential characteristics of the event.

2. **Event Properties Token:** Captures the specific properties or attributes associated with the event. Accounts for variations or details related to the occurrence of the event. Provides the model with additional context about the event beyond its type.

3. **Trace Properties Token:** Signifies the trace-related properties associated with the event. Encompasses information that relates the event to the overall context of the trace. Helps the model understand the role of the event within the broader process flow.

This three-token representation allows the model to learn the inherent relationships between event types, event properties, and trace properties. Importantly, it provides the flexibility for

the model to autonomously discern patterns and associations within the data, aligning with the unsupervised learning nature of DL methods in PPM. By tailoring the tokenisation process to the specific "language of processes", this approach ensures that the model captures the intricacies and nuances present in the sequential data of process logs, facilitating more accurate and context-aware predictions.

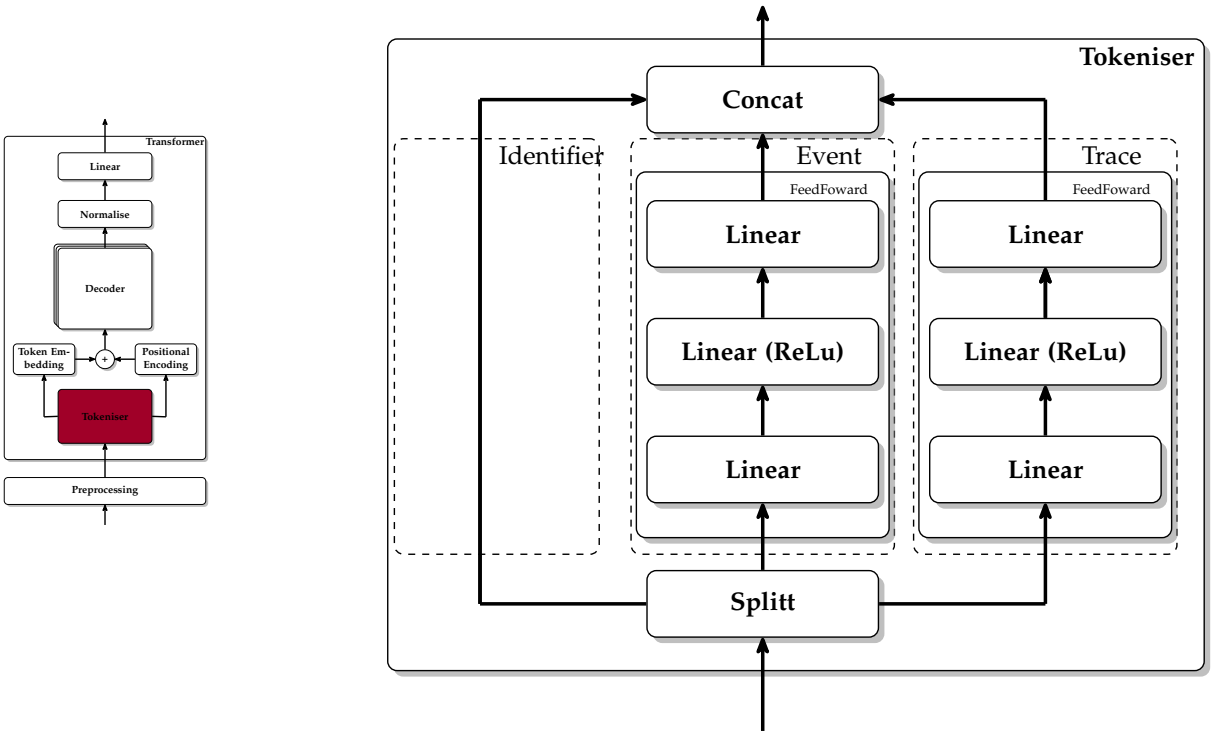


Figure 3.6: Architecture of Tokeniser

The integration of the tokeniser within the larger Transformer architecture is a key design choice, aiming to streamline the processing flow and enhance the model's understanding of the custom "language of processes". The tokeniser's architecture, as illustrated in Figure 3.6, plays a pivotal role in translating the intricate information from event logs into a format that the transformer can effectively comprehend. In this integrated setup, each element describing an event — the Event-Identifier, Event-Properties, and Trace-Properties — undergoes individual tokenisation. These tokenised representations are then concatenated into a unified sequence of tokens.

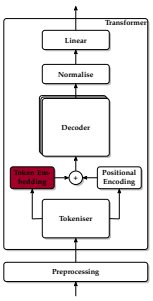
- **Event-Identifier:** This component remains unchanged during tokenisation, preserving the identity of the event. The tokeniser retains the original event identifier, maintaining a direct link to the specific event type.

- **Event-Properties and Trace-Properties:** Both Event-Properties and Trace-Properties undergo tokenisation and subsequent projection using FeedForward layers. The tokeniser incorporates a FeedForward layer for each of these components, comprised of three linear layers. Between the second and third linear layers, a Rectified Linear Unit (ReLU) activa-

tion function is applied to introduce non-linearity. This configuration allows the model to learn complex relationships within the event and trace properties, capturing nuanced patterns and variations (see Section 2.2.1).

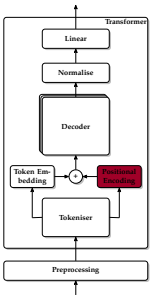
The use of individual FeedForward layers for Event-Properties and Trace-Properties ensures that each type of information undergoes tailored processing. This approach enhances the model's capacity to discern and learn from the diverse aspects of event logs, ultimately contributing to a more effective PPM model.
This approach allows the model to be equipped to handle the intricate details of event logs, and train on them in one model instead of two separate models. This provides a cohesive and unified representation for subsequent stages in the pipeline.

3.2.2.2 Token Embedding



In the designed architecture (Figure 3.1), the tokens generated during the tokenisation process serve as the basis for representing events, event properties, and trace properties. While it is conceivable to use these raw tokens directly, a common practice, as introduced in the original Transformer paper [21], involves employing a learned linear layer with softmax activation to embed the tokens. The reasoning behind using a learned token embedding is to introduce a trainable layer that transforms the discrete token representations into continuous vectors of fixed dimensions. This transformation is decisive for enhancing the model's ability to capture subtle relationships and patterns present in the token sequences. This approach aligns with the design principles of the Transformer architecture, which has been widely adopted in various natural language processing tasks, including language models like GPT-2 [29]. The use of a learned linear layer allows the model to adapt and fine-tune the token representations during the training process.

3.2.2.3 Positional Encoding



The incorporation of positional encoding is a crucial element in the architectural design. As per the original Transformer paper by Vaswani et al. [21], the introduction of positional encoding is vital to provide the model with information about the relative or absolute positions of the tokens in the input sequences. Positional encoding imparts temporal awareness to the model, allowing it to consider the order and timing of events in a sequence. The paper outlines two primary methods for adding positional information to the token embeddings: static encoding and learned encoding. The static approach involves predefining positional encodings based on mathematical functions, such as sine and cosine, depicted in Figure 3.7.

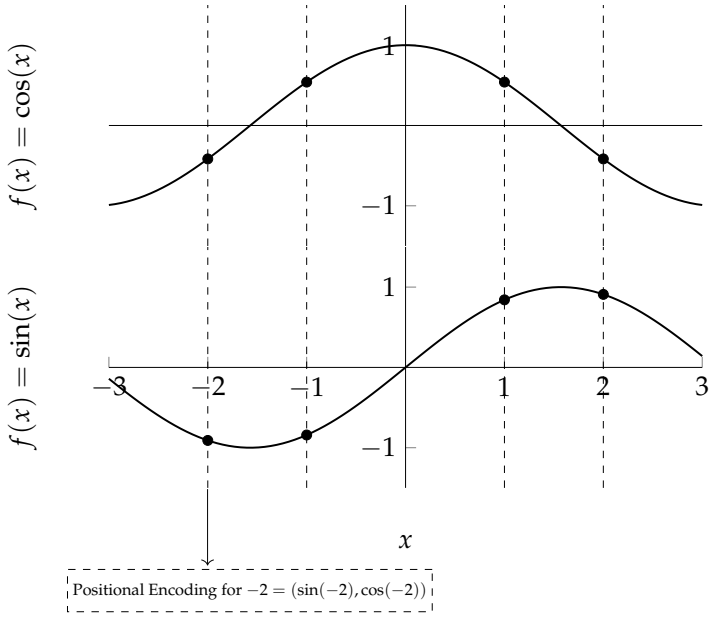


Figure 3.7: Sine and Cosine functions for positional encoding

On the contrary, the learned approach regards positional encodings as trainable parameters, thereby enabling the model to modify them during the training process. Despite comparable results with both approaches, the authors of the original Transformer paper opted for static positional encoding. The rationale behind this decision is the potential benefit of allowing the model to generalise to sequences longer than those encountered during training. This static encoding strategy introduces a certain level of invariance to the absolute position of tokens, enabling the model to handle variable-length sequences more effectively. In the context of PPM, where the goal is to forecast the next event in a sequence, understanding the positional relationships between events is paramount. The positional encoding contributes to the model's ability to discern the order and context of events, facilitating accurate predictions. The functions for positional encoding used in [21] are:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

where:

- pos is the position of a word in the sequence,
- d_{model} is the dimension of the model (which is even),
- i ranges from 0 to $d_{\text{model}}/2$.

The positional encoding function generates a 2D array where the rows correspond to positions and the columns correspond to the encoding dimensions. For each position, half of the dimensions are populated with sine values and the other half with cosine values. For this reason the

embedding dimension needs to be an even number.

3.2.3 Decoder

The Decoder serves as a fundamental building block in the architecture. It is a complex structure composed of multiple identical layers, each containing components for processing and generating sequences, depicted in Figure 3.8. These layers work collaboratively to transform the input data and provide the necessary context for accurate predictions. The decoder is structured with multiple layers stacked on top of each other. This layered approach enables the model to iteratively refine its understanding of the input sequence, gradually extracting relevant features for predictive process mining.

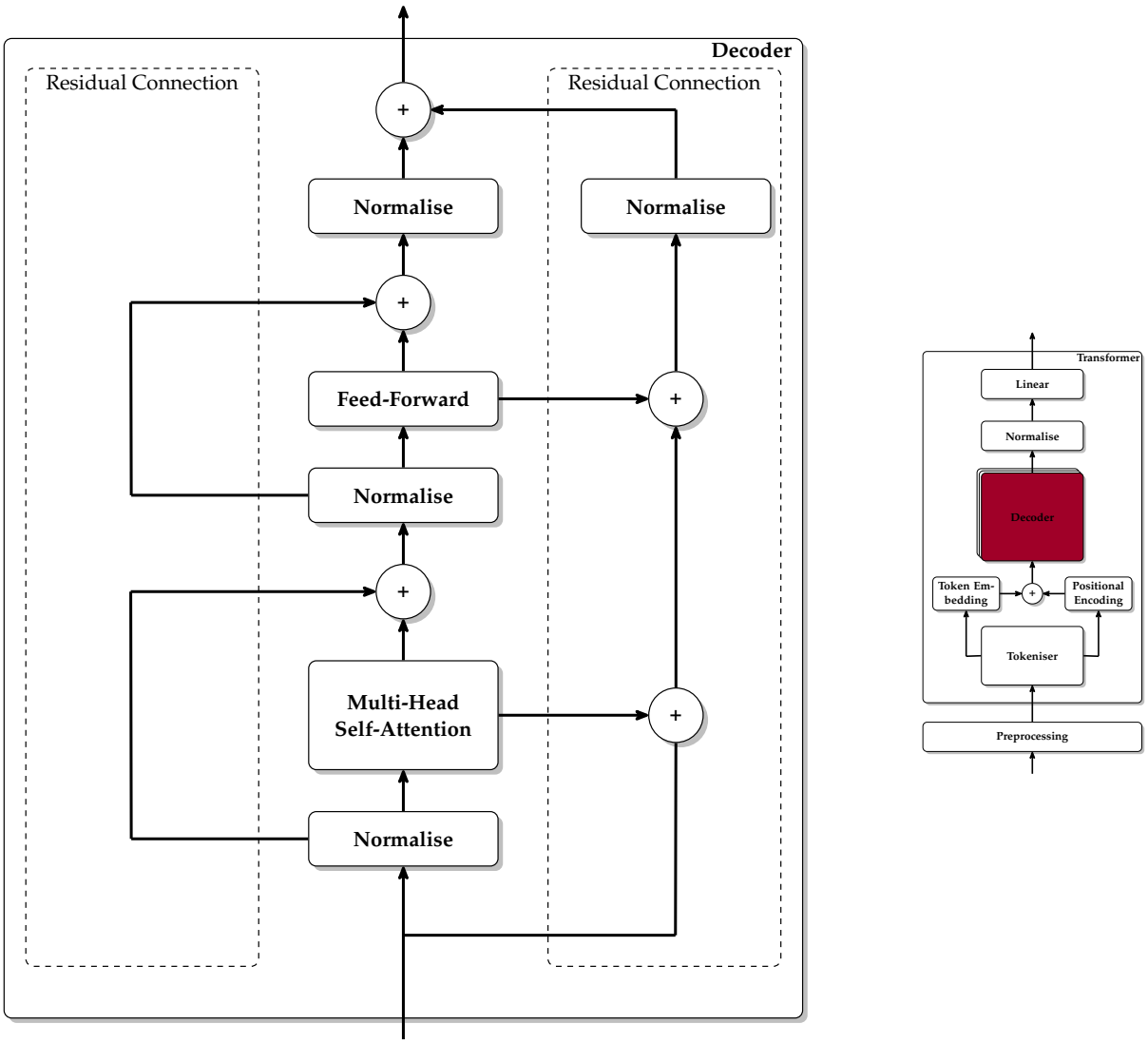


Figure 3.8: Architecture of Decoder-Layer

- **Multi-Head Self-Attention Layer:** This layer allows the model to focus on different parts

of the input sequence simultaneously, capturing intricate relationships between events and their properties.

- **FeedForward Layer:** Responsible for processing the information obtained from the attention layer, the FeedForward layer contributes to capturing complex patterns and relationships within the event sequences.

- **Normalisation Layers:** Applied to stabilise the learning process, normalisation layers ensure that the data is consistently scaled and allows for more efficient training.

- **Residual Connections:** These connections facilitate the flow of information through the layers, mitigating the vanishing gradient problem and enhancing the model's ability to capture long-range dependencies.

3.2.3.1 Multi-Head Self-Attention

Attention mechanisms have proven to be pivotal in capturing intricate relationships within sequences, and the Multi-Head Self-Attention (MHSA) layer plays a crucial role in enhancing the model's ability to understand and weigh different parts of the input sequence effectively.

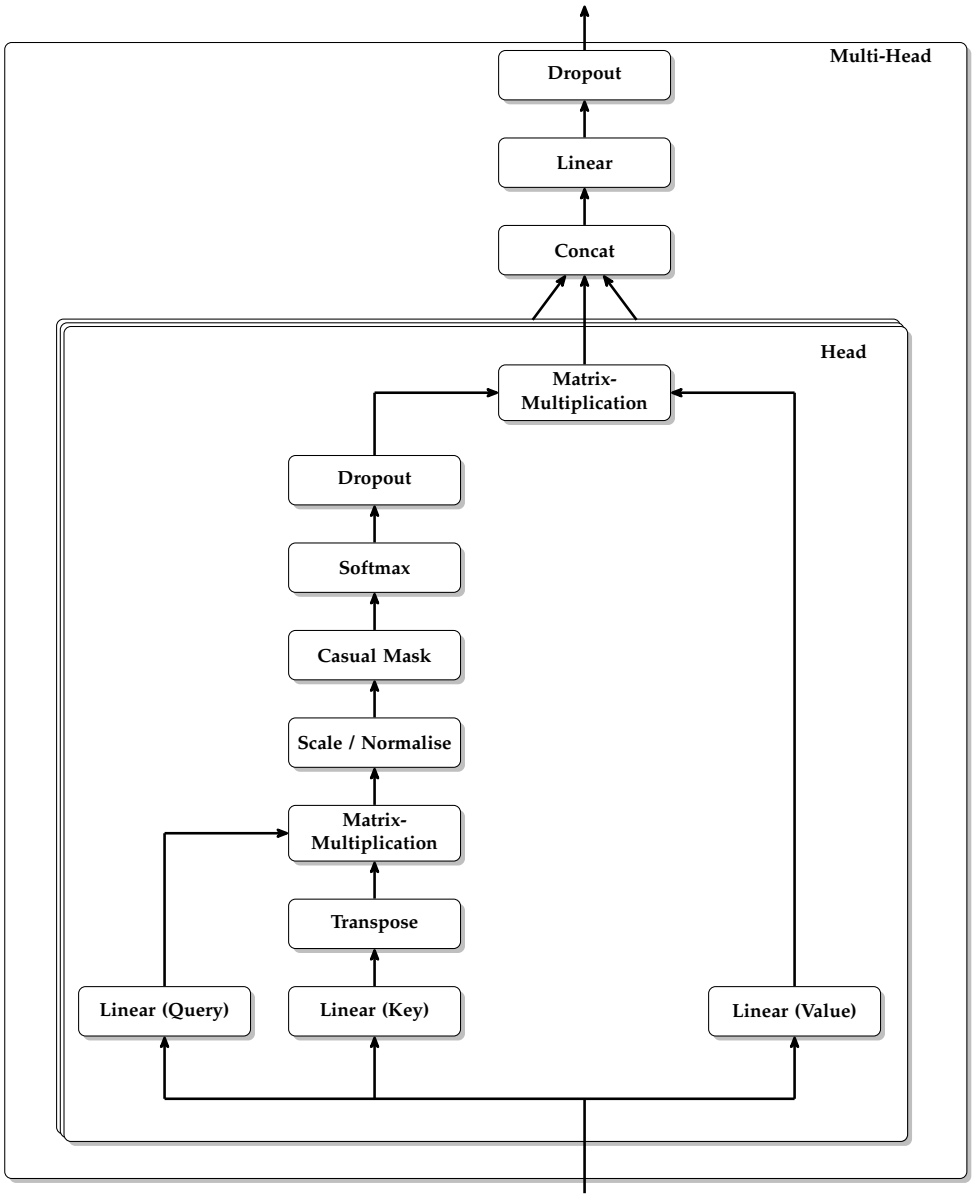


Figure 3.9: Architecture of Multi-Head Self-Attention

In self-attention (Figure 3.9), each token - i.e. every projected feature (channel) of the token - in the sequence is associated with three vectors: The query - representing what the token is looking for -, the key - representing what the token contains -, and the value -which holds the information that will be used in the output. These vectors are linear projections of the input tokens, allowing the model to understand and process the information they carry. The fundamental principle of self-attention is based on matrix multiplication. The query matrix is multiplied with the transposed key matrix, resulting in a set of weights that signify the affinity between different tokens. This operation effectively determines how much attention each token should pay to others. To ensure stable and meaningful attention weights, the result of the matrix multiplication is scaled by the square root of the dimension of keys and queries, as

mentioned in [21]. This scaling normalises the values, preparing them for the softmax activation function, obviating a aggregation of the attention values sharply around the maximum for larger values, a characteristic depicted in Figure 3.10.

The softmax function is defined as follows for a vector $\mathbf{x} := [x_1, x_2, \ldots, x_n]$:

$$\sigma(\mathbf{x})_i := \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Let x be a vector in $\mathbf{R}^n$, where $n \in \mathbf{N}$.

$$\mathbf{x} := \begin{bmatrix} 0.2 \\ -0.1 \\ -0.3 \\ 0.5 \\ 0.3 \end{bmatrix} ; \quad \sigma(\mathbf{x}) \approx \begin{bmatrix} 0.2082 \\ 0.1543 \\ 0.1263 \\ 0.2811 \\ 0.2301 \end{bmatrix} ; \quad \sigma(10 * \mathbf{x}) \approx \begin{bmatrix} 0.0419 \\ 0.0021 \\ 0.0003 \\ 0.8418 \\ 0.1139 \end{bmatrix}$$

Figure 3.10: Characteristic sharpening around the maximum when applying softmax

A causal mask is applied to restrict attention to previous and the token itself only, maintaining the auto-regressive nature of the attention mechanism, effectively making it self-attention. Afterward, the softmax function is applied to obtain a distribution of attention weights. To prevent overfitting and enhance the model's generalisation capabilities, dropout is applied to the attention weights (see Section 3.2.3.5). This dropout is an essential regularisation technique, randomly setting a fraction of the weights to zero during training. The self-attention mechanism is further enriched by employing multiple heads. Each head independently learns different representations of the input sequence. The outputs of these heads are concatenated, providing a more comprehensive and nuanced understanding of the relationships within the sequence. The concatenated outputs undergo a linear transformation, followed by another dropout layer. This transformation allows the model to adaptively weigh the contributions from different heads.

Self-attention is inherently auto-regressive as it conditions each prediction on the preceding tokens in the sequence. This property aligns well with the task of PPM, where understanding the chronological order of events is crucial. Moreover, the parallel computation capabilities of transformers make self-attention highly efficient, allowing the model to process different parts of the sequence simultaneously. While self-attention is widely used, other types of attention mechanisms exist, such as global attention and local attention, which use different inputs for their keys and queries. Self-attention's ability to capture long-range dependencies and relationships across the entire sequence makes it particularly suitable for tasks like PPM, enabling the model to dynamically focus on different aspects of the input sequence, understand complex relationships, and make accurate predictions based on the learned dependencies.

3.2.3.2 FeedForward

The FeedForward layer is a fundamental component in Transformer based architectures, providing non-linear transformations to the model's internal representations. This layer contributes to the overall expressiveness of the model, allowing it to capture intricate dependencies within the sequence and improve its ability to make accurate predictions in the context of PPM. Adapted from the architecture proposed in the original Transformer paper [21], this layer consists of two fully connected linear layers with a Rectified Linear Unit (ReLU) activation function applied between them (Figure 3.11). In the context of PPM, the FeedForward layer plays a crucial role in introducing non-linearity to the model, enabling it to learn complex patterns and relationships within the input sequence. The choice of dimensionality, projecting from 512 to 2048 and then back to 512, used in the original Transformer architecture [21], follows the proportion of 1-4-1, maintaining a balance between expressive capacity and computational efficiency. It is used in this model's FeedForward layer as well.

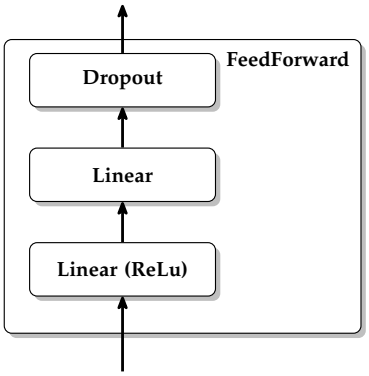


Figure 3.11: Architecture of FeedForward-Network with two Layers

3.2.3.3 Normalisation

Layer normalisation, introduced by Ba et al. [84], a technique that normalises the activities of neurons within a neural network, providing several advantages that contribute to the overall efficiency and stability of the model. One key advantage of layer normalisation is its independence from the batch size. Unlike other normalisation techniques, layer normalisation ensures consistent behaviour during both training and testing, while also being known for its identical computation during training and testing. This independence simplifies the training process and enhances the model's robustness, allowing it to generalise well to different dataset sizes. This characteristic ensures that the normalisation process remains consistent across different phases, preventing issues related to shifting statistics between training and inference. The stability introduced by layer normalisation is crucial in the context of PPM, where model consistency is paramount for reliable predictions. Another notable benefit is its effectiveness in stabilising hidden dynamics within the model, which accelerates the learning process and reduces training time substantially, a crucial factor in complex tasks and deep models.

The placement of layer normalisation layers within the architecture has been a subject of variation in recent developments [85, 86]. The optimal placement is intertwined with the use of

residual connections in the Transformer architecture, a topic that will be explored further in Section 3.2.3.4. The combination of layer normalisation and residual connections enhances the overall performance and training efficiency of the PPM model, providing a stable and scalable solution.

3.2.3.4 Residual Connections

The incorporation of residual connections is a pivotal strategy in the design of the model's architecture. The concept of residual connections, introduced by He et al. [87], addresses the challenges posed by the increasing complexity of neural networks, particularly those with multiple layers. In the original proposal by He et al., residual connections were initially applied to deep neural networks in the context of image recognition. The core idea was to reformulate layers as learning residual functions with reference to the layer input, transforming the learning process. This approach proved effective in easing the training of deep networks by introducing identity mapping, where the original input is connected directly to further layers, effectively bypassing certain layers. This not only streamlined the learning process but also contributed to improved performance, as layers that didn't add valuable projections were skipped.

In the context of transformers, which form the basis of the PPM architecture proposed in this thesis, residual connections find application in both the multi-head self-attention and the feed-forward layers. However, the placement of normalisation layers in relation to residual connections has evolved since the original proposal by Vaswani et al. [21]. Xiong et al. [85] delved into the discussion of the optimal placement of normalisation layers concerning residual connections. They categorised architectures into Post-LN (Layer Normalisation after residual connections) and Pre-LN (Layer Normalisation before residual connections). The former aligns with the original proposal by Vaswani et al., while the latter introduces normalisation before residual connections. More recently, Xie et al. proposed a novel architectural approach called ResiDual, aiming to leverage the advantages of both Post-LN and Pre-LN architectures [86]. They observed that Post-LN architectures suffer from gradient vanishing issues, hindering efficient training. On the other hand, Pre-LN architectures may lead to representation collapse, potentially losing information before applying an operation. The ResiDual approach seeks to address both challenges, preventing gradient vanishing and representation collapse. By combining elements from both Post-LN and Pre-LN architectures, Xie et al. claim to achieve a more robust and efficient training process.

In the proposed architecture, as depicted in Figure 3.8, the ResiDual approach has been adopted. This choice is driven by the desire to maintain the benefits of residual connections while mitigating potential issues related to gradient vanishing and representation collapse. This consideration of architectural components contributes to the overall effectiveness and stability of the model in handling PPM tasks.

3.2.3.5 Dropouts

The incorporation of dropout, a regularisation method introduced by Srivastava et al. [88], plays a crucial role in creating a deep neural network. The primary objective of dropout is to

prevent units within a neural network from co-adapting excessively, thereby mitigating the risk of overfitting. The overfitting problem arises when a model learns the training data too well, capturing noise and idiosyncrasies specific to the training set, but failing to generalise effectively to unseen data. Dropout addresses this challenge by introducing a form of randomness during the training process. The mechanism of dropout involves randomly "dropping out" units and their connections during each training iteration. This random dropout forces the network to operate as if it were a collection of sub-networks, each learning specific features independently. By doing so, dropout promotes the development of more robust and generalised representations within the model. The application of dropout within the proposed architecture serves to enhance its overall performance. By preventing units from relying too heavily on specific features or relationships within the training data, dropout encourages the model to learn a more diverse and adaptive set of features. This diversity is helping the model in its ability to handle variations and complexities present in different PM scenarios.

The effectiveness of dropout in improving the performance of neural networks has led to its widespread adoption in various domains. It contributes to the creation of a model that not only learns from the intricacies of the training data but also generalises well to unforeseen process instances. The strategic integration of dropout within a PPM model aligns with the best practices in deep learning regularisation. This ensures that the model remains resilient, avoids overfitting pitfalls, and demonstrates robust predictive capabilities across a spectrum of real-world process scenarios.

3.3 Usage and Workflow

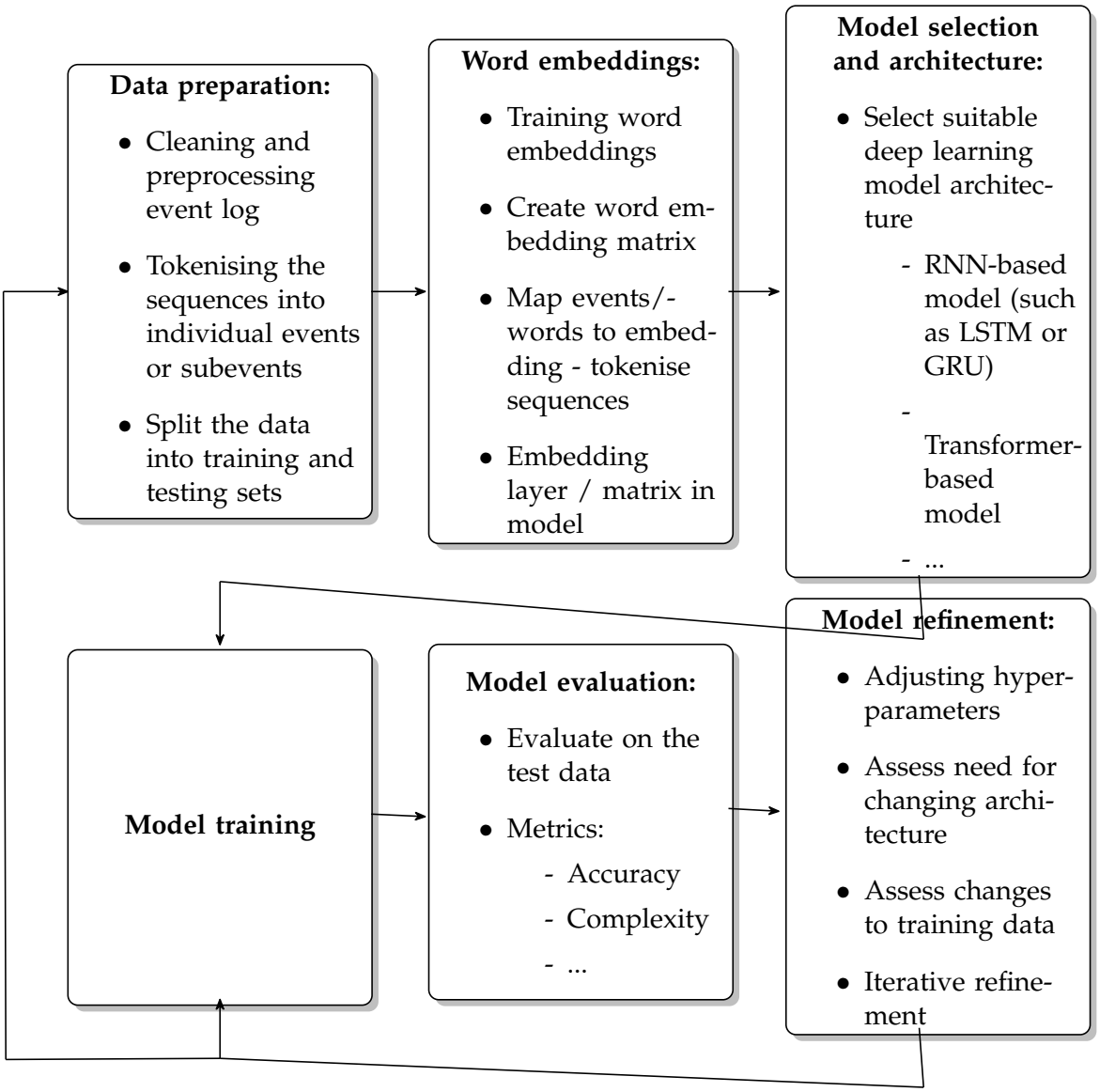


Figure 3.12: Concept for creation of a PPM-Model

4 Implementation

In this section, we provide a concise overview of the implementation of the theoretical design and concepts discussed in Chapter 3. Each subsection presents code snippets with brief explanations and outlines any deviations from the original concept. The goal is to bridge the gap between theory and practical implementation, highlighting key design decisions and their impact on the actual code.
The subsequent subsections will cover specific aspects of the implementation, providing insights into the code and discussing any adjustments from the initial theoretical design.

4.1 Structure

The repository for this thesis is kept in a simple manner. It contains the processed event logs (the orignal event logs are published as part of the BPI Challenges [1]).

```
Thesis
|   model.py
|   preprocessor.py
|   train.py
|
+---datasets
|   \---processor_results
|       +---bpi_17
|       |   +---maps
|       |   |       event_map.json
|       |   |       event_props_map.json
|       |   |       trace_props_map.json
|       |   |
|       |   \---trace
|       |           events_props.json
|       |           traces.json
|       |           traces_props.json
|       |
|       \---bpi_19
|           +---maps
|           |       event_map.json
|           |       event_props_map.json
|           |       trace_props_map.json
|           |
|           \---trace
|                   events_props.json
|                   traces.json
|                   traces_props.json
|
```

[1] https://data.4tu.nl/search?search=BPI+Challenge

```
\---eval
        17_dataset_1000_iters_100_seqL_10_batchsize_every_20_metrics.json
        19_dataset_1000_iters_100_seqL_10_batchsize_every_20_metrics.json
```

Listing 4.1: Structure of repository

Beside the data and translation map, there are three Python scripts. "model.py" contains the code for the model, "preprocessor.py" contains the code for preprossessing a XES event log, and "train.py" is the script, that contains the training loop, as well as the evaluation. Finally, there is a evaluation directory, that contains log files from the training.

4.2 Used Frameworks and Software

The implementation of the model described in this thesis was carried out using the Python programming language. Python was selected for its extensive support in the ML community and compatibility with popular DL frameworks such as TensorFlow and PyTorch. Python, known for its readability and versatility, facilitated the development process. Although the implementation was done using Python version 3.11, any version equal to or higher than 3.9 should be suitable for running the code. This choice of programming language ensures broad accessibility and compatibility with existing libraries and tools. For the construction of the PPM model, TensorFlow, a widely used open-source ML framework, was employed. The version utilised for this implementation was TensorFlow 2.15. TensorFlow provides comprehensive support for building and training DL models, making it a natural choice for this project. The decision to use TensorFlow aligns with the framework's popularity in the ML and DL communities. Its extensive documentation, active community support, and robust set of functionalities contribute to a seamless development experience. TensorFlow's compatibility with both CPUs and GPUs allows for flexibility in deploying models across various computing environments. Nonetheless, the entire implementation can be accomplished using Pytorch, which appears to be a popular option for coding Transformer based DL model based on transformers. Overall, the combination of Python as the programming language and TensorFlow as the ML framework provided a solid foundation for the implementation of the model. These choices aim to enhance readability, maintainability, and accessibility, essential factors in the development and future iterations of the model.

4.3 Preprocessing

The primary objectives of preprocessing are to extract relevant information from event logs, encode the data into a format suitable for model input, and create mapping files for interpretation and prediction translation. One of the key tasks in preprocessing is the extraction of pertinent details from event logs. This includes identifiers, event properties, and trace properties. Categorical data, such as event identifiers, is represented using integer encodings. Continuous data is encoded as floating-point numbers. The encoding is consistent across different logs, ensuring uniformity for the model's training. The XES-Standard serves as the input format. While the XES-Standard allows for arbitrary amounts of attributes per event, only the globally defined attributes will be accounted for, since they are ensured to be defined on all elements with a default value. Since XES is XML based, Pythons build in library ElementTree was used

to process it.
The task of the preprocessing can be divided into two main steps: Extract translation maps for the encoding for the classifier of events, the event attributes that are globaly defined, and the trace Properties that are globaly defined. And secondly, the creation of the trace in the encoded format, i.e. the data is transformed into a numerical format compatible with the model. The translation-maps, as well as the encoded traces, are saved in JSON-files. These files serve a dual purpose. First, they provide a human-readable record of how each part of the log was encoded. This transparency facilitates understanding and verification of the encoding process. Second, the mapping files act as a reference for translating model predictions back into interpretable events, traces, and their respective properties. For instance, if the model predicts a numerical encoding for an event identifier, the mapping file can be consulted to map this encoding back to the original identifier. This functionality is essential for making the model's output interpretable and applicable in real-world scenarios.

4.3.1 Extracting Translation Maps

```
global_trace_attributes = {}
for global_trace in log.findall('.//global[@scope="trace"]'):
    for attribute in global_trace:
        if attribute.tag in [ 'date', 'int', 'float', 'boolean']:
            global_trace_attributes[attribute.get('key')] = {
                'value': attribute.get('value'),
                'type': attribute.tag
            }
        elif attribute.tag in [ 'string', 'id']:
            global_trace_attributes[attribute.get('key')] = {
                'value': attribute.get('value'),
                'type': attribute.tag,
                'map': {
                    attribute.get('value'): 0
                }
            }

sorted_global_trace_attributes = {k: global_trace_attributes[k] for k in sorted(
    global_trace_attributes)}
```

Listing 4.2: Preprocessing Implementation - Extraction of all globaly defined Trace-Properties

The code snippet in Listing 4.2 outlines the procedure for extracting these trace attributes and initialising the translation map. The extraction process involves iterating through all <global> tags in the log with a scope of "trace." These tags in XES-Standard contain global attributes that apply to the entire trace. The translation map, implemented as a Python dictionary, is initialised with default values for common attribute types such as 'date,' 'int,' 'float,' 'boolean,' 'string,' and 'id.' However, it's important to note that the implementation focuses on these attribute types due to their prevalence, and other attribute types, which were considered in the concept, are not considered in this context. For attributes like 'id' and 'string,' additional steps are taken to initialise lookup tables (Python dictionaries). These lookup tables are necessary for tracking the peculiarities of these attributes, providing a mapping between the original attribute values and their numerical representations. They are filled with mapping information during encoding of the traces. This is particularly important for maintaining interpretability

and translating model predictions back into human-understandable traces.
The extraction of the initial Event-Properties-Map is generated in a similar manner, as depicted in Listing 4.3.

```
global_event_attributes = {}
for global_event in log.findall('.//global[@scope="event"]'):
    for attribute in global_event:
        if attribute.tag in [ 'date', 'int', 'float', 'boolean']:
            global_event_attributes[attribute.get('key')] = {
                'value': attribute.get('value'),
                'type': attribute.tag
            }
        elif attribute.tag in [ 'string', 'id']:
            global_event_attributes[attribute.get('key')] = {
                'value': attribute.get('value'),
                'type': attribute.tag,
                'map': {
                    attribute.get('value'): 0
                }
            }

sorted_global_event_attributes = {k: global_event_attributes[k] for k in sorted(
    global_event_attributes)}
```

Listing 4.3: Preprocessing Implementation - Extraction of all globaly defined Event-Properties

Furthermore, an additional step not explicitly mentioned in the initial concept is introduced in the code. The attributes are sorted alphabetically. This ensures a consistent order, especially when using lookup tables, as some programming languages, including Python prior to version 3.7, do not inherently preserve the order of dictionary elements. This alphabetical sorting step enhances the reproducibility of the encoding process across programming environments.
The decision to focus on specific attribute types and simplify the translation process is driven by the goal of proving the conceptual feasibility of the model, rather than providing an exhaustive pre-trained model. The chosen attributes cover common scenarios encountered in PM, making the model adaptable to a wide range of practical use cases.

```
classifier_key = None
classifier_default = 0
classifier = log.find('.//classifier')
if classifier is not None:
    classifier_key = classifier.get('keys')
    classifier_default = classifier.get('value')

event_identifier_map = {
    'EOS': 0,
    # Default
    classifier_default : 1,
}

for event in log.findall('.//event'):
    event_classifier = event.find(f".//string[@key='{classifier_key}']")
    if event_classifier is not None:
        e_val = event_classifier.get('value')
        if e_val not in event_identifier_map.keys():
```

```
19                event_identifier_map[e_val] = len(event_identifier_map)
```

Listing 4.4: Preprocessing Implementation - Extraction of all Event-Classifier and creation of look-up for Event-Classifier pecularities

The last translation map generated during preprocessing is specifically for the event classifier or identities. This translation map serves as a lookup table for the event classes, providing a mapping between the classifier peculiarities and their numerical representations, which are named identifier in the code. The code snippet in Listing 4.2 showcases the extraction of the event classifier and the subsequent iteration through all events to create a mapping for the peculiarities of the classifier. This involves associating each event with its corresponding classifying attribute and generating a lookup table that captures the unique characteristics of each event class. The lookup ensures that the model can comprehend the diverse event classes present in the log data.

Listing 4.5, 4.6 , and 4.7 provide example mappings, produced with the code.


```
{
    "EOS": 0,
    "null": 1,
    "SRM: Created": 2,
    "SRM: Complete": 3,
    "SRM: Awaiting Approval": 4,
    "SRM: Document Completed": 5,
    "SRM: In Transfer to Execution Syst.": 6,
    "SRM: Ordered": 7,
    "SRM: Change was Transmitted": 8,
    "Create Purchase Order Item": 9,
    "Vendor creates invoice": 10,
    "Record Goods Receipt": 11,
    "Record Invoice Receipt": 12,
    "Clear Invoice": 13,
    "Record Service Entry Sheet": 14,
    "SRM: Transfer Failed (E.Sys.)": 15,
    "Cancel Goods Receipt": 16,
    "Vendor creates debit memo": 17,
    "Cancel Invoice Receipt": 18,
    "Change Delivery Indicator": 19,
    "Remove Payment Block": 20,
    "SRM: Deleted": 21,
    "Change Price": 22,
    "Delete Purchase Order Item": 23,
    "SRM: Transaction Completed": 24,
    "Change Quantity": 25,
    "Change Final Invoice Indicator": 26,
    "SRM: Incomplete": 27,
    "SRM: Held": 28,
    "Receive Order Confirmation": 29,
    "Cancel Subsequent Invoice": 30,
    "Reactivate Purchase Order Item": 31,
    "Update Order Confirmation": 32,
    "Block Purchase Order Item": 33,
    "Change Approval for Purchase Order": 34,
    "Release Purchase Order": 35,
    "Record Subsequent Invoice": 36,
```

```
    "Set Payment Block": 37,
    "Create Purchase Requisition Item": 38,
    "Change Storage Location": 39,
    "Change Currency": 40,
    "Change payment term": 41,
    "Change Rejection Indicator": 42,
    "Release Purchase Requisition": 43
}
```

Listing 4.5: Example of Classifier-Mapping

```
{
    "Cumulative net worth (EUR)": {
        "value": "0.0",
        "type": "float"
    },
    "User": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "batch_00": 1,
            "user_000": 2,
            "NONE": 3,
            "user_001": 4,
            "user_002": 5,
            "user_004": 6,
            "user_005": 7,
            "user_003": 8,
            // ...
        }
    },
    "concept:name": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "SRM: Created": 1,
            "SRM: Complete": 2,
            "SRM: Awaiting Approval": 3,
            "SRM: Document Completed": 4,
            "SRM: In Transfer to Execution Syst.": 5,
            "SRM: Ordered": 6,
            "SRM: Change was Transmitted": 7,
            "Create Purchase Order Item": 8,
            "Vendor creates invoice": 9,
            "Record Goods Receipt": 10,
            "Record Invoice Receipt": 11,
            "Clear Invoice": 12,
            "Record Service Entry Sheet": 13,
            "SRM: Transfer Failed (E.Sys.)": 14,
            "Cancel Goods Receipt": 15,
            "Vendor creates debit memo": 16,
            "Cancel Invoice Receipt": 17,
            "Change Delivery Indicator": 18,
            "Remove Payment Block": 19,
            "SRM: Deleted": 20,
```

```
            "Change Price": 21,
            "Delete Purchase Order Item": 22,
            "SRM: Transaction Completed": 23,
            "Change Quantity": 24,
            "Change Final Invoice Indicator": 25,
            "SRM: Incomplete": 26,
            "SRM: Held": 27,
            "Receive Order Confirmation": 28,
            "Cancel Subsequent Invoice": 29,
            "Reactivate Purchase Order Item": 30,
            "Update Order Confirmation": 31,
            "Block Purchase Order Item": 32,
            "Change Approval for Purchase Order": 33,
            "Release Purchase Order": 34,
            "Record Subsequent Invoice": 35,
            "Set Payment Block": 36,
            "Create Purchase Requisition Item": 37,
            "Change Storage Location": 38,
            "Change Currency": 39,
            "Change payment term": 40,
            "Change Rejection Indicator": 41,
            "Release Purchase Requisition": 42
        }
    },
    "org:resource": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "batch_00": 1,
            "user_000": 2,
            "NONE": 3,
            "user_001": 4,
            "user_002": 5,
            "user_004": 6,
            "user_005": 7,
            "user_003": 8,
            // ...
        }
    },
    "time:timestamp": {
        "value": "1970-01-01T00:00:00.000Z",
        "type": "date"
    }
}
```

Listing 4.6: Example of Event-Property-Mapping

```
{
    "Company": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "companyID_0000": 1,
            "companyID_0001": 2,
            "companyID_0002": 3,
```

```
            "companyID_0003": 4
        }
    },
    "Document Type": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "EC Purchase order": 1,
            "Standard PO": 2,
            "Framework order": 3
        }
    },
    "GR-Based Inv. Verif.": {
        "value": "false",
        "type": "boolean"
    },
    "Goods Receipt": {
        "value": "false",
        "type": "boolean"
    },
    "Item": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "00001": 1,
            "00002": 2,
            "00003": 3,
            "00004": 4,
            "00005": 5,
            "00006": 6,
            "00007": 7,
            "00008": 8,
            // ...
        }
    },
    "Item Category": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "3-way match, invoice before GR": 1,
            "3-way match, invoice after GR": 2,
            "Consignment": 3,
            "2-way match": 4
        }
    },
    "Item Type": {
        "value": "",
        "type": "string",
        "map": {
            "": 0,
            "Standard": 1,
            "Service": 2,
            "Consignment": 3,
            "Subcontracting": 4,
```

```
            "Third-party": 5,
            "Limit": 6
        }
    },
    "Name": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "vendor_0000": 1,
            "vendor_0001": 2,
            "vendor_0002": 3,
            "vendor_0003": 4,
            "vendor_0004": 5,
            "vendor_0005": 6,
            "vendor_0006": 7,
            "vendor_0007": 8,
            // ...
         }
    },
    "Purch. Doc. Category name": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "Purchase order": 1
        }
    },
    "Purchasing Document": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "2000000000": 1,
            "2000000001": 2,
            "2000000002": 3,
            "2000000003": 4,
            "2000000004": 5,
            "2000000005": 6,
            "2000000006": 7,
            "2000000007": 8,
            // ...
        }
    },
    "Source": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "sourceSystemID_0000": 1
        }
    },
    "Spend area text": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
```

```
            "CAPEX & SOCS": 1,
            "Marketing": 2,
            "Enterprise Services": 3,
            "Workforce Services": 4,
            "Others": 5,
            "Logistics": 6,
            "": 7,
            "Sales": 8,
            "Real Estate": 9,
            "Packaging": 10,
            "Additives": 11,
            "Specialty Resins": 12,
            "Solvents": 13,
            "Trading & End Products": 14,
            "Pigments & Colorants": 15,
            "Titanium Dioxides": 16,
            "Latex & Monomers": 17,
            "Commodity Resins": 18,
            "Spend Area Unidentified": 19,
            "Chemicals & Intermediates": 20,
            "Energy": 21
        }
    },
    "Spend classification text": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "NPR": 1,
            "OTHER": 2,
            "": 3,
            "PR": 4
        }
    },
    "Sub spend area text": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "Facility Management": 1,
            "Marketing Support Services": 2,
            "Digital Marketing": 3,
            "Office Supplies": 4,
            "HR Services": 5,
            "Advertising": 6,
            "ICT Software": 7,
            "Laboratory Supplies & Services": 8,
            // ...
        }
    },
    "Vendor": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "vendorID_0000": 1,
            "vendorID_0001": 2,
```

```
            "vendorID_0002": 3,
            "vendorID_0003": 4,
            "vendorID_0004": 5,
            "vendorID_0005": 6,
            "vendorID_0006": 7,
            "vendorID_0007": 8,
            // ...
        }
    },
    "concept:name": {
        "value": "UNKNOWN",
        "type": "string",
        "map": {
            "UNKNOWN": 0,
            "2000000000_00001": 1,
            "2000000001_00001": 2,
            "2000000002_00001": 3,
            "2000000003_00001": 4,
            "2000000003_00002": 5,
            "2000000003_00003": 6,
            "2000000004_00001": 7,
            "2000000005_00001": 8,
            // ...
        }
    }
}
```

Listing 4.7: Example of Trace-Property-Mapping

4.3.2 Encoding of the Trace to Numerical Values

The task of encoding the trace into numerical values is performed, adhering to the rules outlined in Table 3.1. The implementation involves a helper function, "value_map," which is detailed in Listing 4.8. This function not only returns the encoded value but also updates the entries in the respective mappings of trace or event properties. A divergence to the conceptual translation is the scaling of dates by dividing it by the approximate time of a year. A step to prevent too high spikes in the encoding, that could cause unwanted distortion in the model, which could increase the number of iteration of training needed.

```
def value_map(key, type, value, *,  event_scope=True):
    map = sorted_global_event_attributes if event_scope else
    sorted_global_trace_attributes

    match type:
        case 'string' |  'id':
            if map[key]['map'].get(value) is None:
                res = map[key]['map'][value] = len(map[key]['map'])
            else:
                res = map[key]['map'][value]
            return res
        case 'date':
            unix_timestamp = datetime.datetime.strptime(value, "%Y-%m-%dT%H:%M:%S.%f
    %z").timestamp()
            # Scale timestamp
```

```
            res = unix_timestamp / 31_557_600_000
            return res
        case 'int' | 'float':
            return float(value)
        case 'boolean':
            return 1 if value in ['true', 'True', 'TRUE'] else -1
        case _:
            return
```

Listing 4.8: Preprocessing Implementation - Mapping function, for conversion to numerical values

Furthermore, the implementation contains a nested loop, depicted in Listing 4.9, iterating through each trace and their attributes, as well as each trace's events and their respective attributes. Notably, the Property-Mapping for each trace or event is initialised with a copy of the default value mapping in lines 2 and 12 in Listing 4.9. While these mappings are not shallow copies, the second-level nested dictionary is not copied, but its reference is passed. This ensures that the map is only held in memory once while also guaranteeing that every element always has access to the most current map.

```
for trace in log.findall('trace'):
    trace_props = {k : v.copy() for k,v in sorted_global_trace_attributes.items()}
    events_per_trace_encoded = []
    event_id_per_trace = [] # Trace:[ id, ... ]

    for element in trace:
        key = element.get('key')
        if key in trace_props.keys():
            trace_props[key]['value'] = element.get('value')

        if element.tag == 'event':
            event_props =  {k : v.copy() for k,v in sorted_global_event_attributes.
    items()}

            for attribute in element:
                e_key = attribute.get('key')
                if e_key in event_props.keys():
                    event_props[e_key]['value'] = attribute.get('value')

                if e_key == classifier_key:
                    event_id_per_trace.append(event_identifier_map[attribute.get('
    value')])

            event_props_encoded = [value_map(k, v['type'], v['value'], event_scope=
    True) for k,v in event_props.items()]

            event_props_encoded.extend([0] * (PROP_DIM - len(event_props_encoded)))

            events_per_trace_encoded.append(event_props_encoded)

    eos_identfier = event_identifier_map['EOS']
    eos_event_props = [0] * PROP_DIM

    event_id_per_trace.append(eos_identfier)
    result_identifier += event_id_per_trace

    events_per_trace_encoded.append(eos_event_props)
```

```
    result_event_props += events_per_trace_encoded

    trace_props_encoded = [value_map(k, v['type'], v['value'], event_scope=False)
    for k,v in trace_props.items()]

    trace_props_encoded.extend([0] * (PROP_DIM - len(trace_props_encoded)))
    trace_props_encoded = [trace_props_encoded] * len(event_id_per_trace)

    result_trace_props += trace_props_encoded
```

Listing 4.9: Preprocessing Implementation - Extraction of and encoding of Trace (Identifier, Event-Properties, Trace-Properties

A significant deviation from the original concept is the decomposition of the three encodings that make up each event in each trace into three different series. This is done due to the characteristics of tensors in TensorFlow, which only allow uniformly shaped objects. Additionally, each of the traits that make up the event is translated into a token separately in the next step, making it easier if the input is composed of three sequences, each representing one trait (event-id, event-properties, trace-properties). The order of the events in each produced sequence remains the same, ensuring that each event is reconstructable with all its traits. This decomposition simplifies the subsequent tokenisation step, facilitating the creation of meaningful token sequences.

4.4 Transformer

The implementation of the Transformer model is a fundamental component of the overall project. The model is encapsulated within a class named "Transformer," as depicted in Listing 4.10. This class serves as a wrapper for the various layers that constitute the Transformer architecture. The "Transformer" class encapsulates four essential layers:
Input Tokeniser: Responsible for converting the input data, which is in the form of tuples representing events in a trace, into a format suitable for further processing.
Sequence Encoding: This layer takes the tokenised input and encodes it into a continuous vector space. It essentially transforms the input sequences into a format suitable for the subsequent decoding steps.
Decoder Block: This layer contains multiple stacked decoder layers. Each decoder layer is responsible for processing the encoded input sequences, capturing intricate patterns, and enhancing the model's understanding.
Output Projection: The final layer of the model, responsible for projecting the output of the decoder block into the desired format. In this case, it transforms the continuous vector space back into sequences of events, maintaining the structure of the original input.

```
class Transformer(tf.keras.Model):
    def __init__(self,vocab_size ,embedding_dim=1_000, n_head=6, n_layers=6,
    batch_size=40, dropout=0.15, prop_dim=1_000, sequence_length=100, ffwd_dim=None)
    :
        super().__init__()
        # B = batch_size
        T = sequence_length
        C = embedding_dim

        self.input_tokeniser = InputTokeniser(prop_dim=prop_dim)
```

```
        self.sequence_encoding = SequenceEncoding(embedding_dim=C, batch_size=
    batch_size, vocab_size=vocab_size, sequence_length=sequence_length)
        self.decoder_block = tf.keras.Sequential([
            DecoderLayer(embedding_dim=C, n_head=n_head, dropout=dropout, ffwd_dim=
    ffwd_dim)
            for _ in range(n_layers)
        ])
        self.output_projection = OutputProjection(prop_dim=prop_dim, vocab_size=
    vocab_size)

    def call(self, x):
        x = self.input_tokeniser(x)
        x = self.sequence_encoding(x)
        x = self.decoder_block(x)
        x = self.output_projection(x)

        return x
```

Listing 4.10: Transformer Implementation

During the forward pass (call method), the input data is sequentially passed through these layers. The output of one layer serves as the input to the next, facilitating the transformation of the input data through the entire model. This sequential processing allows the model to learn complex patterns and relationships within the input data, ultimately producing meaningful predictions. The organisation of the layers within the "Transformer" class adheres to the conceptual design outlined in Chapter 3. The model's architecture follows the principles of the Transformer, with an input tokenisation step, encoding of the sequences, multiple decoder layers, and a final output projection.

4.4.1 Tokeniser

The implementation of the tokeniser is encapsulated within the class named "InputTokeniser," as depicted in Listing 4.11. The tokenisation process is divided into three distinct pipelines, each handling a specific component of the input data:

- The identifiers, representing the types of events in the input data, are into tensors. They are projected into a continuous vector space, introducing a level of abstraction. This is a crucial step, as it allows the model to distinguish between different types of events during processing.
- The event properties are already continuous vectors. Hence, they are projected into a continuous vector space, that has the same shape as the identfier-output. In doing so, we convert all tokens into the same shape, enabling the model to learn this three token per event representation of the "process language".
- Similar to event properties, trace properties are also fed through a identical, but seperate pipeline, leading to an output of the same shape.

```
class InputTokeniser(tf.keras.layers.Layer):
    def __init__(self, prop_dim=1_000):
        super().__init__()
        self.event_tokeniser = tf.keras.Sequential([
            tf.keras.layers.Dense(prop_dim * 4),
```

```
            tf.keras.layers.Dense(prop_dim * 0.5, activation='relu'),
            tf.keras.layers.Dense(1),
            tf.keras.layers.Lambda(lambda x: tf.squeeze(x, axis=-1)),
        ])
        self.trace_tokeniser = tf.keras.Sequential([
            tf.keras.layers.Dense(prop_dim * 4),
            tf.keras.layers.Dense(prop_dim * 0.5, activation='relu'),
            tf.keras.layers.Dense(1),
            tf.keras.layers.Lambda(lambda x: tf.squeeze(x, axis=-1)),
        ])

    def call(self, x):
        identifier, event_props, trace_props = zip(*x)
        event_props =  tf.cast(tf.convert_to_tensor(event_props), dtype=tf.float32)
        trace_props =  tf.cast(tf.convert_to_tensor(trace_props), dtype=tf.float32)

        identifier_token = tf.cast(tf.convert_to_tensor(identifier), dtype=tf.
    float32)
        event_token = self.event_tokeniser(event_props)
        trace_token = self.trace_tokeniser(trace_props)

        stacked = tf.stack([identifier_token, event_token, trace_token], axis=-1)
        B, T, _ = stacked.shape
        tokens = tf.reshape(stacked, [B, 3*T])

        return tokens
```

Listing 4.11: Tokeniser Implementation

After the tokenisation of each component, the resulting tokens are stacked together, maintaining the original order of the events. The concatenation of these tokens effectively increases the length along the time dimension by a factor of three, aligning with the model's expectation of one input sequences in the subsequent layers.
The tokeniser's implementation aligns with the conceptual design outlined in Chapter 3, where each component of the input (identifier, event properties, trace properties) undergoes a specific transformation, contributing to the model's ability to understand and process the input data effectively. The three pipelines work cohesively to prepare the input for the subsequent encoding and decoding steps in the Transformer model.

4.4.2 Encoding

The encoding layer, as depicted in Listing 4.12, serves as a bridge between the tokenisation stage and the subsequent layers of the Transformer model. This layer encompasses both token embedding and positional encoding, ensuring that the model can effectively process the tokenised input. The first step in the encoding process involves vectorising the tokens representing each event. As each event is represented by three tokens, the Embedding Layer from TensorFlow is utilised to create a lookup table for these tokens. The Embedding Layer assigns a unique vector to each token based on its integer value. In this case, the integer values correspond to the identifiers of the events. Since the first token in each triple is an integer and the other two tokens are of type float, a projection step is necessary to ensure uniformity in the dimensions. The non-integer tokens are projected using a Linear Layer, transforming them into

the same shape as the vectorised integer tokens. This step aligns the embeddings and prepares them for further processing. After the vectorisation and projection, all three embeddings (one for the integer token and two for the non-integer tokens) are combined into a single tensor. The resulting tensor has a unified shape of (Batch size, Context length, Embedding Dimension), where the Context length represents the length of the sequences after tokenisation.

```
class SequenceEncoding(tf.keras.layers.Layer):

    def __init__(self, batch_size=40, vocab_size=100, embedding_dim=1_000,
    sequence_length=100):
        super().__init__()
        B = batch_size
        T = sequence_length
        C = embedding_dim
        self.token_i_emb = tf.keras.layers.Embedding(vocab_size, C)
        self.token_e_emb = tf.keras.layers.Dense(T*C, use_bias=False)
        self.token_t_emb = tf.keras.layers.Dense(T*C, use_bias=False)

    def call(self, x):
        i = x[:,0::3]
        e = x[:,1::3]
        t = x[:,2::3]

        i = tf.cast(i, dtype=tf.int32)
        i = self.token_i_emb(i)
        B, T, C = i.shape

        e = self.token_e_emb(e)
        t = self.token_t_emb(t)

        e = tf.reshape(e, (B,T,C))
        t = tf.reshape(t, (B,T,C))

        stacked = tf.stack([i, e, t], axis=2)
        x = tf.reshape(stacked, (B, 3*T, C))

        x += SequenceEncoding.positional_encoding(3*T, C)

        return x

    @staticmethod
    def positional_encoding(sequence_length: int, dimension: int) -> tf.Tensor:
        assert dimension % 2 == 0, \
            f"Embedding-Dimension must be an even number, but was {dimension}."

        dimension = dimension // 2

        positions = np.arange(sequence_length)[:, np.newaxis]

        dimensions = np.arange(dimension)[np.newaxis, :] / dimension

        angle_rates = 1 / (10000 ** dimensions)
        angle_rads = positions * angle_rates

        pos_encoding = np.concatenate(
            [np.sin(angle_rads), np.cos(angle_rads)],
```

```
            axis=-1
        )

        return tf.cast(pos_encoding, dtype=tf.float32)
```

Listing 4.12: Encoding Implementation

To incorporate the temporal order of events within a sequence, positional encoding is added to the combined embeddings. The positional encoding is introduced as a static method of the layer class, ensuring that the positional information is added consistently across all instances. The choice of using static positional encoding, as discussed in Section **??**, aligns with the original Transformer architecture proposed by Vaswani et al **??**. The implementation of the encoding layer seamlessly integrates token embedding and positional encoding, producing a tensor that retains both the semantic information from the tokenised input and the temporal relationships between events.

4.4.3 Decoder-layer

The decoder layer, as presented in Listing 4.13, encapsulates essential components such as the attention mechanism, normalisation layers, residual connections, and a feedforward layer. At the core of the decoder layer is the multi-head self-attention mechanism. This mechanism allows the model to focus on different parts of the input sequence simultaneously. The attention weights are calculated by linearly projecting the queries, keys, and values, applying scaled dot-product attention, and then concatenating and linearly projecting the outputs.

```
class DecoderLayer(tf.keras.layers.Layer):

    def __init__(self, embedding_dim, n_head, ffwd_dim=None, dropout=0.15):
        super().__init__()
        # B = batch_size
        # T = sequence_length
        C = embedding_dim

        head_size = embedding_dim // n_head

        self.norm_x = tf.keras.layers.LayerNormalization()
        self.mhsa = tf.keras.layers.MultiHeadAttention(
            num_heads=n_head,
            key_dim=head_size,
            dropout=dropout,
            use_bias=False,
        )
        self.add_x_mhsa = tf.keras.layers.Add()
        self.add_norm_mhsa = tf.keras.layers.Add()
        self.norm_mhsa = tf.keras.layers.LayerNormalization()
        self.ffwd = FeedForward(embedding_dim=embedding_dim, ffwd_dim=ffwd_dim,
    dropout=dropout)
        self.add_x_ffwd = tf.keras.layers.Add()
        self.add_norm_ffwd = tf.keras.layers.Add()
        self.norm_ffwd = tf.keras.layers.LayerNormalization()
        self.norm_residual = tf.keras.layers.LayerNormalization()
        self.add_x_residual = tf.keras.layers.Add()

```

```
    def call(self, x):
        norm_x = self.norm_x(x)
        attention = self.mhsa(
            query=norm_x,
            key=norm_x,
            value=norm_x,
            use_causal_mask=True)
        resi_1 = self.add_x_mhsa([x, attention])
        resi_2 = self.add_norm_mhsa([attention, norm_x])
        norm_mhsa = self.norm_mhsa(resi_2)
        forward = self.ffwd(norm_mhsa)
        resi_3 = self.add_x_ffwd([resi_1, forward])
        resi_4 = self.add_norm_ffwd([forward, norm_mhsa])
        norm_ffwd = self.norm_ffwd(resi_4)
        norm_residual = self.norm_residual(resi_3)
        output = self.add_x_residual([norm_residual, norm_ffwd])

        return output
```

Listing 4.13: Decoder Implementation

The residual connection, as well as the normalisation, are employed as described in Chapter 3, helping to stabilise the learning process and accelerates training. The decoder layer incorporates a feedforward neural network, visible in Listing 4.14, consisting of two fully connected linear layers with a ReLU activation function in between.

```
class FeedForward(tf.keras.layers.Layer):
    def __init__(self, embedding_dim=1_000, ffwd_dim=None, dropout=0.15):
        super().__init__()
        # B = batch_size
        # T = sequence_length
        C = embedding_dim

        ffwd_dim = ffwd_dim or 4 * C

        self.ffwd = tf.keras.Sequential([
            tf.keras.layers.Dense(ffwd_dim, activation='relu'),
            tf.keras.layers.Dense(C),
            tf.keras.layers.Dropout(dropout),
        ])

    def call(self, x):
        return self.ffwd(x)
```

Listing 4.14: FeedForward Implementation

4.4.4 Output Projection

The output projection layer, illustrated in Listing 4.15, serves as the final step in the Transformer model, responsible for converting the model's predictions into a format that aligns with the original input structure. This layer decomposes the predicted token into three distinct parts, corresponding to the identifier, event properties, and trace properties. The identifier token, representing the event type, is projected into logits using a linear layer. The linear transformation aims to map the input vector to a space where each dimension corresponds to a different event

type in the vocabulary. This allows for the calculation of probabilities through a softmax activation, ultimately yielding the predicted event type. Token associated with event properties, which was previously embedded and processed within the model, are projected into a vector representing the predicted values for each property dimension. This projection enables the model to generate output that aligns with the structure of the original event properties. Similar to the event properties token, the trace properties token is projected into a vector, predicting the values for each property dimension, ensuring that the model's predictions for trace properties are in the original format.

```
class OutputProjection(tf.keras.layers.Layer):
    def __init__(self, prop_dim=1_000, vocab_size=100):
        super().__init__()
        self.norm = tf.keras.layers.LayerNormalization()
        self.identifier = tf.keras.layers.Dense(vocab_size)
        self.event_props_projection = tf.keras.layers.Dense(prop_dim)
        self.trace_props_projection = tf.keras.layers.Dense(prop_dim)

    def call(self, x):
        x = self.norm(x)
        identity = x[:,::3,:]
        event_props = x[:,1::3,:]
        trace_props = x[:,2::3,:]

        identity = self.identifier(identity)
        event_props = self.event_props_projection(event_props)
        trace_props = self.trace_props_projection(trace_props)

        identity_unstacked = tf.unstack(identity, axis=0)
        event_props_unstacked = tf.unstack(event_props, axis=0)
        trace_props_unstacked = tf.unstack(trace_props, axis=0)

        combined_list = [(id_batch, ev_batch, tr_batch) for id_batch, ev_batch,
    tr_batch in zip(identity_unstacked, event_props_unstacked, trace_props_unstacked
    )]

        return combined_list
```

Listing 4.15: Output Projection Implementation

The three projected values—identifier logits, event properties, and trace properties—are stacked back together to form a list. Each element in this list corresponds to a triple for each input batch. The elements of the output triples are tensors with shapes: (Context length, Vocabulary size) for the identifier, (Context length, Property Dimension) for event properties, and (Context length, Property Dimension) for trace properties. The output projection layer, through its tailored projections, aligns the model's output with the expected structure of the input, allowing for meaningful predictions in terms of event types and associated properties.

4.5 Training

The training of the Transformer model involves several key components and follows a structured process. This overview, while concise, provides insights into the key steps of the training implementation.

The training process is organised into epochs, each consisting of a series of training iterations. An epoch represents a pass through the training dataset. In each iteration, the model undergoes forward and backward passes to adjust its parameters. The training function, depicted in Listing 4.16, encapsulates the essential steps for each training iteration.

```
1
2  def training(model=None, epochs=10, iters_per_epoch=500, batch_size=60,
       sequence_length=200, learning_rate=3e-4, save_path='./models', model_name='
       transformer'):
3      model = model if model else Transformer(
4          vocab_size=vocab_size,
5          embedding_dim=1_000,
6          n_head=6,
7          n_layers=6,
8          batch_size=batch_size,
9          dropout=0.1,
10         prop_dim=100,
11         sequence_length=sequence_length,
12         ffwd_dim=None
13     )
14
15     optimizer = tf.keras.optimizers.AdamW(learning_rate=learning_rate)
16     x, _ = generate_batch(batch_size=batch_size, sequence_length=sequence_length,
       split='train')
17     model(x)
18     variables = model.trainable_variables
19     optimizer.build(variables)
20     loss_fn = weighted_loss_fn
21     loss_col = []
22
23     for epoch in range(epochs):
24
25         for iter in tqdm(range(iters_per_epoch), desc=f"Epoch {epoch+1}", unit="
       iteration"):
26
27             xb, yb = generate_batch(batch_size=batch_size, sequence_length=
       sequence_length, split='train')
28             with tf.GradientTape() as tape:
29                 y_pred = model(xb)
30
31                 loss = loss_fn(yb, y_pred, step=iter)
32
33             gradients = tape.gradient(loss, model.trainable_variables)
34             gradients_and_variables = [(grad, var) for grad, var in zip(gradients,
       model.trainable_variables) if grad is not None]
35             optimizer.apply_gradients(gradients_and_variables)
36
37         losses = estimate_current_loss(
38             model,
39             iters=int(max((iters_per_epoch / 10), 1)),
40             batch_size=batch_size,
41             sequence_length=sequence_length)
42
43         loss_col.append(losses)
44
45     model.save(f"{save_path}/model_{model_name}.keras")
46
```

```
47     return model, loss_col
```

Listing 4.16: A function for training the mode on a predifined dataset

The process begins with creating batched data, which involves organising the input data and their respective labels into batches. These batches are then fed into the model to obtain predictions. Following predictions, the loss for the current iteration is computed. The loss serves as a measure of the disparity between the predicted values and the actual labels. This value is needed for guiding the optimisation process. Subsequently, the gradients are calculated using backpropagation, and the optimiser utilises these gradients to update the model's parameters. This iterative optimisation process incrementally refines the model's ability to make accurate predictions. The described steps are repeated for each iteration within an epoch. After completing all iterations in an epoch, the model is evaluated using the custom loss estimator. This process is reiterated for the specified number of epochs.
The training function provides a foundational structure for optimising the Transformer model. The use of epochs, iterations, and a custom loss metric ensures a systematic approach to improving the model's performance over successive training cycles. The efficacy of the training process is further explored and analysed in detail in Chapter 5.

The batch generation function, detailed in Listing 4.17, facilitates the creation of batched input data and their corresponding labels, allowing the model to learn from sequential data in a systematic and controlled manner. The function takes a parameter for the context length, allowing flexibility in defining the size of the context window for each batch. This context length parameter influences how much historical information the model considers when making predictions. The batch generation function can be applied to both training and validation datasets, ensuring uniformity in the data preparation process for both training and evaluating the model. The core of the batch generation process involves randomly selecting indices within the permissible range. These indices determine the starting positions of the sequences in the log.

```
1
2  def generate_batch(batch_size=60, sequence_length=200, split='train'):
3      (data_i, data_e, data_t) = (train_i, train_e, train_t) if split == 'train' else
       (val_i, val_e, val_t)
4
5      idx = tf.random.uniform(shape=(batch_size,), maxval=len(data_i) -
       sequence_length, dtype=tf.int32)
6
7      x_i = tf.stack([tf.cast(tf.convert_to_tensor(data_i[i:i+sequence_length]), dtype
       =tf.int32) for i in idx])
8      x_e = tf.stack([tf.convert_to_tensor(data_e[i:i+sequence_length]) for i in idx])
9      x_t = tf.stack([tf.convert_to_tensor(data_t[i:i+sequence_length]) for i in idx])
10
11     x = [*zip(x_i, x_e, x_t)]
12
13     y_i = tf.stack([tf.cast(tf.convert_to_tensor(data_i[(i+1):(i+1)+sequence_length
       ]), dtype=tf.int32) for i in idx])
14     y_e = tf.stack([tf.convert_to_tensor(data_e[(i+1):(i+1)+sequence_length]) for i
       in idx])
15     y_t = tf.stack([tf.convert_to_tensor(data_t[(i+1):(i+1)+sequence_length]) for i
       in idx])
16
```

```
    y = [*zip(y_i, y_e, y_t)]

    return x, y
```

Listing 4.17: Function that creates a batched data chunk of inputs and their labels fro either the train or validation splitt

The random selection is done uniformly, ensuring an equal likelihood for each possible index to be chosen. Using the randomly chosen indices, sequences are extracted from the entire log. For each index, the input data is extracted starting from that index and continuing until the specified sequence length (context length). This process effectively creates batches of sequences, with each sequence representing a context window for the model. Labels for the batches are generated in a similar manner to the input data. However, to create labels that correspond to the predictions the model should make, the indices for labels are shifted to the right by one position. This shift aligns the labels with the next element in the sequence, as the model's objective is to predict the subsequent elements based on the given context. By randomising the selection of indices and incorporating a customisable context length, this batch generation function introduces variability and adaptability into the training process. It ensures that the model encounters diverse contexts during training, contributing to its ability to generalise and make accurate predictions across different scenarios. The effectiveness of this batch generation strategy is assessed in Chapter 5.

The custom loss function, outlined in Listing 4.18, plays a pivotal role in training the Transformer model due to the unique nature of the tasks it performs. The Transformer model is designed to handle two distinct tasks simultaneously during training. The first task involves predicting the next token in the sequence, which is treated as a classification problem. The second task entails directly predicting the values for the properties of both the event and the trace, represented by the subsequent two tokens in the sequence. For the classification task of predicting the event type (token logits), the cross-entropy loss is employed. Cross entropy is a suitable choice for classification problems, as it measures the difference between the predicted probability distribution and the true distribution of the target class. The prediction of continuous values for event and trace properties is treated as a regression problem. To assess the model's accuracy in predicting these values, the Mean Absolute Error (MAE) loss is utilised. MAE calculates the average absolute difference between the predicted values and the ground truth, providing a measure of how well the model predicts continuous variables.

```

def weighted_loss_fn(y_true, y_pred, weights=(1, 0.01, 0.01), step=0):
    loss_fn_i = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
    mse_loss_fn = tf.keras.losses.MeanAbsoluteError()

    i_pred, e_props_pred, t_props_pred = [tf.convert_to_tensor(t) for t in zip(*
    y_pred)]
    i_true, e_props_true, t_props_true = [tf.convert_to_tensor(t) for t in zip(*
    y_true)]

    loss_i = loss_fn_i(i_true, i_pred)
    loss_e_props = mse_loss_fn(e_props_true, e_props_pred)
    loss_t_props = mse_loss_fn(t_props_true, t_props_pred)

```

```
    if step % 2 == 0:
        return weights[0] * loss_i
    else:
        return weights[1] * loss_e_props + weights[2] * loss_t_props
```

Listing 4.18: Implementation of a custom loss function for the model

The challenge arises from the fact that these two types of losses - cross entropy for classification and MAE for value predictions - operate on different scales. To address this, the custom loss function alternates between calculating the cross-entropy loss and the MAE loss for each iteration. This alternating approach ensures that both types of losses are considered throughout the training process. The function includes dynamic weighting for the two types of losses, as weights are adjusted based on the ratio of classification tokens to value prediction tokens in the sequence. This dynamic weighting helps maintain balance and prevents dominance of one type of loss over the other. TensorFlow automatically tracks the variables and their associated gradients during the training process. This feature is leveraged to alternate between the two types of loss calculations seamlessly, allowing for the employment of different loss functions on different parts of the model (i.e. in the output-projection), making the custom loss function tailored to the dual nature of tasks performed by the Transformer model.

```

def mean_accuracy(y_true, y_pred):
    i_pred, _, _ = [tf.convert_to_tensor(t) for t in zip(*y_pred)]
    i_true, _, _ = [tf.convert_to_tensor(t) for t in zip(*y_true)]

    i_true = tf.cast(i_true, dtype=tf.int32)

    correct_predictions = tf.equal(i_true, tf.cast(tf.argmax(i_pred, -1), dtype=tf.int32))
    accuracy = tf.reduce_mean(tf.cast(correct_predictions, tf.float32))

    return accuracy

def estimate_current_loss(model, iters=200, batch_size=60, sequence_length=200):
    result = {
        'id': {'loss': {}, 'accuracy': {}},
        'props': {}
    }
    model.trainable = False

    for split in ['train', 'validate']:
        losses = []
        accuracies = []
        for _ in range(iters):
            xb, yb = generate_batch(batch_size=batch_size, sequence_length=sequence_length, split=split)
            y_pred = model(xb)
            loss = weighted_loss_fn(yb, y_pred, step=0)
            losses.append(loss)

            accuracy = mean_accuracy(yb, y_pred)
            accuracies.append(accuracy)

        result['id']['loss'][split] = np.mean(losses)
        result['id']['accuracy'][split] = np.mean(accuracies)
```

```
34
35      for split in ['train', 'validate']:
36          losses = []
37          for _ in range(iters):
38              xb, yb = generate_batch(batch_size=batch_size, sequence_length=
    sequence_length, split=split)
39              y_pred = model(xb)
40              loss = weighted_loss_fn(yb, y_pred, step=1)
41              losses.append(loss)
42
43          result['props'][split] = np.mean(losses)
44
45      model.trainable = True
46
47      return result
```

Listing 4.19: A Function that estimates the current loss for a set of iterations after each epoch on the training and the validation data

The loss estimator, outlined in Listing **??**, offers real-time insights into the model's performance during training. The loss estimator is designed to provide immediate feedback on the model's performance during training, as it calculates the mean loss and accuracy over approximately 10% of the iterations within each epoch. It computes the mean loss for both the identifier (classification task) and the properties (value prediction task), as well as the mean accuracy for the identifier. This distinction allows monitoring the model's proficiency in both tasks separately. By estimating the loss periodically during an epoch, practitioners gain insights into how well the model is adapting to the training data and whether it is overfitting (i.e. the training loss decreases, while the validation loss stays the same or increases). This live feedback aids in making timely adjustments to hyperparameters, architecture, or other aspects of the training process and served as a guideline for chosing which hyperparameters result in the most performant model.

In the implemented Transformer model, several hyperparameters play a role in shaping the model's architecture and governing the training process. The embedding dimension, denoted as "embedding_dim", determines the size of the embedding vectors for tokens. As described in Chapter 3 the embedding dimension is also intertwined with the hyperparameter feedforward dimension. While it can be defined separately, it defaults to being derived as four times the embedding dimension. This dimensionality determines how complex (in terms of dimensions of the representing vector) the model represents and processes input tokens. "n_head" specifies the number of attention heads in the Multi-Head Attention mechanism. The "n_layers" hyperparameter defines the depth of the Transformer model by specifying the number of layers in the decoder block. Dropout sets the probability of dropping out units during training, while the learning rate determines the step size during the optimisation process. It influences how quickly or slowly the model adapts to the training data. Determining how much historical information the model considers when making predictions, the "sequence_length" hyperparameter defines the length of the input sequences (context) used during training.

5 Evaluation

In assessing the performance of the Transformer model, it's crucial to consider the nature of the task and choose evaluation metrics that align with the model's objectives. Unlike traditional classification tasks where accuracy is a standard metric, evaluating a LLMs or a Generative Pre-trained Transformer (GPT) model involves nuanced considerations. Even though the proposed model is not a LLM, the metrics used to evaluate it, have to be chosen in alignment with the type of tasks its supposed to solve.

5.1 Metrics

While accuracy is a common metric for classification tasks, its application to Language Models, especially those designed for generative tasks, may not fully capture the model's capabilities. A wide range of test for evaluating LLM's such as BART [64] or Chat-GPT [22], have evolved around the topic of LLm solving NLP tasks (e.g. [89]), along with competitions for handing in problems that these type of models perform bad in such as "Inverse Scaling Prize" [1] , which are then used for evaluating the capabilities of the model [71].

In the context of this thesis, where the model is tasked with understanding and generating sequences of events, accuracy alone might not be sufficient. Language Models are often evaluated on their ability to generate coherent, contextually relevant, and informative sequences. Metrics assess how well the generated sequences match reference sequences or capture relevant information. Given that the model is designed to understand the intricacies of processes and predict the next event in a sequence, a meaningful evaluation should focus on the model's ability to generate plausible and contextually appropriate event predictions, regarding their type, as well as their characteristics, i.e. its properties. This involves assessing the quality of the generated output in terms of relevance to the input and the relation to its properties. As for very specific processes and task, very specific metrics may be more informative. In the evaluation of the model, especially when dealing with logs that the model has not been trained on, assessing accuracy becomes a challenging yet critical aspect.

The model's capability to predict the next event in a sequence for an unseen process provides valuable insights into its generalisation abilities. For the identifier tokens, which determine the type of event, accuracy is a relevant metric. This metric measures how often the model correctly predicts the next event type in the sequence. The accuracy is calculated as the ratio of correctly predicted event types to the total number of predictions. This provides a clear understanding of the model's proficiency in identifying the correct event type. Unlike identifier tokens, property tokens represent continuous values, and their prediction is akin to time

[1] https://github.com/inverse-scaling/prize

series forecasting. Two common metrics for such evaluations are Mean Squared Error (MSE) and Mean Absolute Error (MAE). MSE calculates the average of the squared differences between predicted and actual values. It penalises larger errors more heavily, making it sensitive to outliers in the prediction. MAE calculates the average of the absolute differences between predicted and actual values. It provides a more straightforward measure of the average prediction error, regardless of the direction (overestimation or underestimation). Both MAE and MSE offer a quantitative assessment of the model's accuracy in predicting continuous values. Lower values for these metrics indicate better performance. Combining accuracy for identifier tokens and MAE or MSE for property tokens could provide a holistic overall evaluation. It captures the model's ability not only to identify the correct event type but also to predict the continuous properties of the events accurately. Aligning with the approach in related work, as seen in **??**, where accuracy is used for event prediction and MAE for remaining execution time prediction, demonstrates consistency in the evaluation strategy within the field.

5.2 Limitations of the Evaluation

The training of ML models is a resource-intensive process, and the success of the training phase significantly impacts the model's performance. The training conduct for this thesis faced limitations in hardware resources, particularly the available RAM and the type of GPU, posed significant challenges throughout the training cycle. The Intel Core i7 with 4 cores and 16 GB RAM, that have been available for training the model, presented hard constraints for model training, especially with the complex architectures. Besides the CPU, the available NVIDIA GeForce GTX 1070 GPU was only of limited use, due to TensorFlow's discontinuation of native support for Windows, we had to resort to using the Windows Subsystem for Linux (WSL) for GPU training. This, however, came at the cost of reduced available RAM (around 6GB) due to the constraints imposed by WSL. To overcome the limitations of GPU training in WSL, we opted to run the training on the CPU. While this approach allowed to proceed with training, it came at the expense of significantly extended training times, reaching multiple hours for even smaller models (e.g. one epoch with a batch size of 4, 500 iterations, and a sequence length of 50 - which can be considered a small training - took more than 12 hours. The resource limitations affected various aspects of the training process, such as context length, batch sizes, and model layer dimensions. These compromises can impact the model's ability to capture intricate patterns and dependencies in the data.

Acknowledging the challenges we sought for a delay in the submission date, however, to this date, there has been no response from the examinations board, which lead to a time crunch for extensive model training. Given the challenges in the training process, the evaluation of the model's capabilities becomes inherently speculative, since the model evaluated is of small scale and, while trained with all resources available, can be considered barely trained. Without a well-trained model, calculating accuracy or assessing performance on unseen logs becomes unlikely, making it difficult to impossible to differ between inaccuracy caused by these training constraints and the actual capabilities of the model. Hence, the following evaluation is solely based on the indications the available model provides.

5.3 Results

Despite the limitations, we experimented with different loss functions and different weights for the combination of the loss functions during training. The weighted and alternating loss function, as described in Section 4.5, showed the best results in terms of reducing the overall loss, in the limited number of passes that have been done.
The results presented in this section arised from training the model on the datasets provided for the BPI Challenge in 2017 and 2019 [2]. On each dataset, the model has been trained for 1000 iterations and evaluated every 20 iterations. The used hyperparameter are:

- Batch size: 20
- Context length: 100
- Embedding dimension: 1000
- Number of heads: 6
- Number of layers: 6
- Dropout rate: 10%
- Feedforward dimension: 4000
- Optimiser: AdamW
- Leearning rate: $3 * 10^{-4}$
- Train-Validation-Split: 80% - 20%

The results of the model trained on the dataset form 2017 are depicted in Figure 5.1 - 5.4, those of the model trained on the dataset from 2019 in Figure 5.5 - 5.8. Besides the CCross-Entropy and the Mean-Absolute-Error as a measurement for the loss, the Accuracy (regarding the identifier) has been monitored.

[2] https://data.4tu.nl/search?search=BPI+Challenge

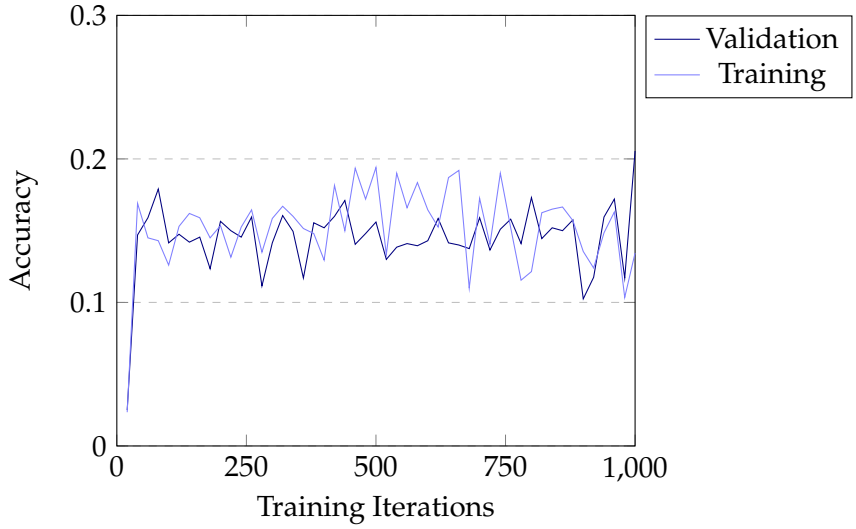


Figure 5.1: Accuracy of the model over the training iteration on the BPI-Challenge Dataset from 2017

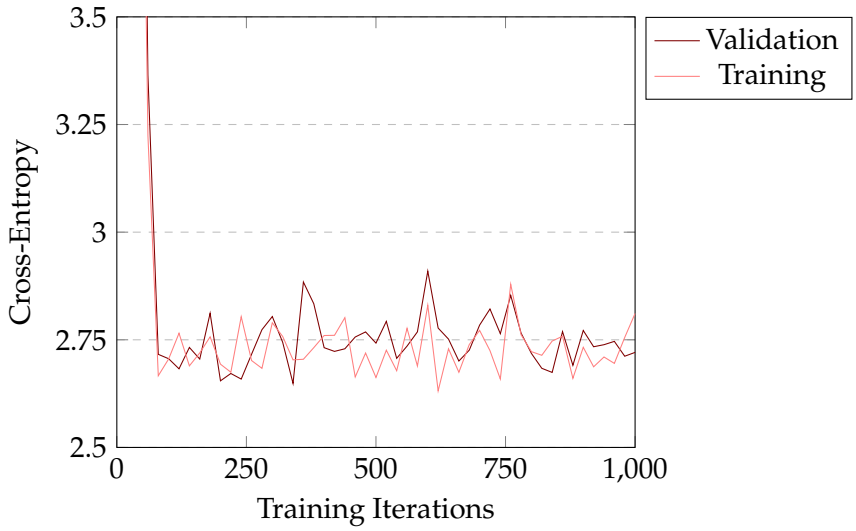


Figure 5.2: Cross-Entropy of the model over the training iteration on the BPI-Challenge Dataset from 2017

In both training instances, the accuracy exhibits a similar pattern. Initially, during the first 50 iterations, there is a modest increase in accuracy, reaching around 15%. However, beyond this point, the accuracy fluctuates. Notably, the accuracy levels of both the training and validation sets remain relatively consistent. This suggests that the model is not excessively memorising the training data (overfitting), as the accuracy on unseen validation data aligns with that on the training data. There are discernible differences in the accuracy fluctuations between the two training instances. In the training on the 2017 dataset, the accuracy fluctuates less, ranging between approximately 10In contrast, the training on the 2019 dataset shows more fluctuations, varying between 0A notable observation is that, regardless of the dataset used, the accuracy never surpasses 20This limitation suggests that the model might face challenges in capturing

the intricacies of the underlying processes, and further optimisation may be needed.

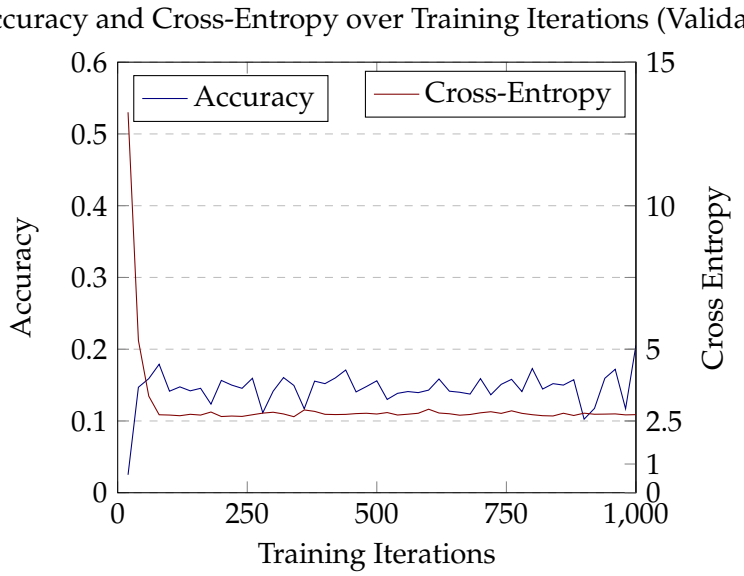


Figure 5.3: Performance of the model over the training iteration on the BPI-Challenge Dataset from 2017

Similar to accuracy, cross-entropy exhibits fluctuations during both training instances. These fluctuations, particularly in the 2019 dataset, suggest challenges in stabilising the model's predictions and capturing the intricacies of the underlying processes. In both training scenarios, the cross-entropy initially shows a decreasing trend during the first approximately 50 iterations. This decline signifies an initial learning phase where the model adapts to the training data. However, beyond this phase, the cross-entropy starts to fluctuate, and more importantly does not decrease any further. Particularly the fluctuations in cross-entropy are more pronounced in the training on the 2019 dataset compared to the 2017 dataset. The varying nature of the 2019 data might pose additional challenges for the model in maintaining a consistent level of prediction quality.

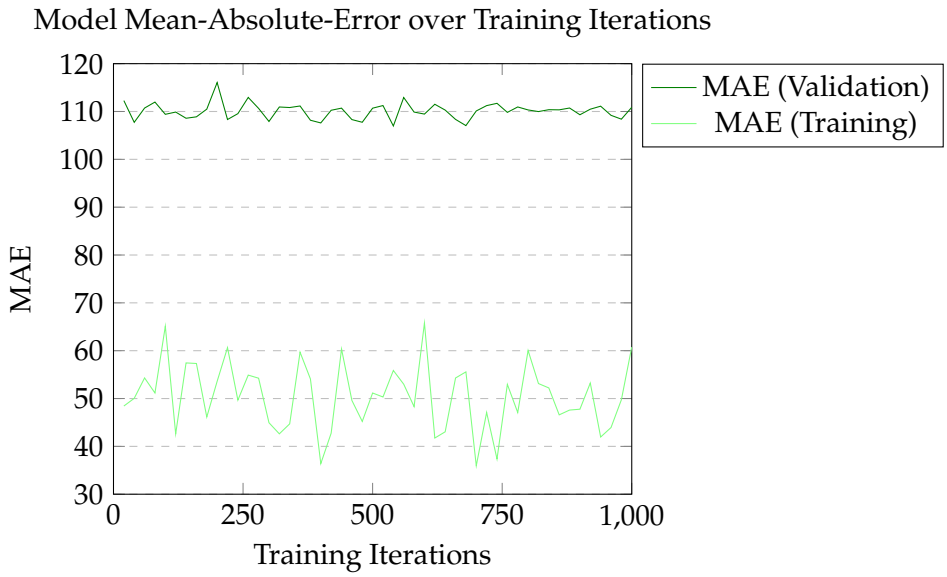


Figure 5.4: Performance of the model over the training iteration on the BPI-Challenge Dataset from 2017

The observed patterns in cross-entropy align with those in accuracy, especially in the 2017 dataset. Figure 5.3 illustrates a similar trend in both accuracy and cross-entropy, indicating a potential correlation between the two metrics. However, this correlation is less evident in the 2019 dataset (Figure 5.7), suggesting more complex learning dynamics.

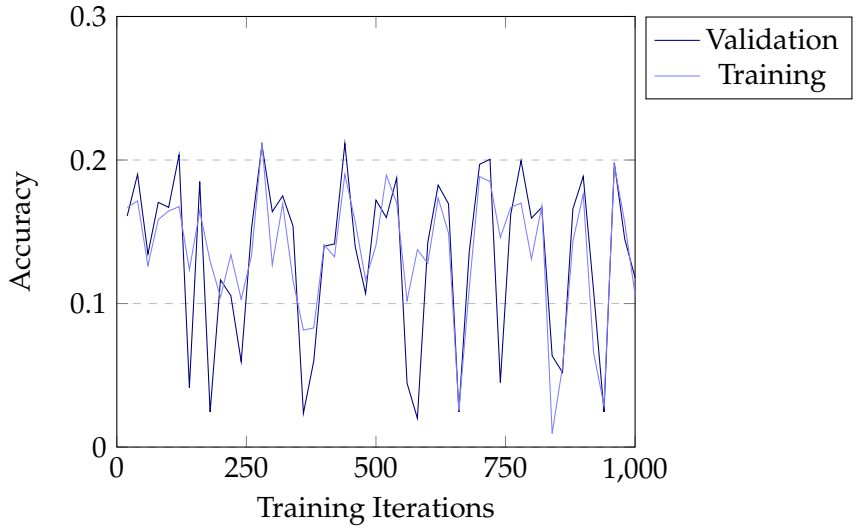


Figure 5.5: Accuracy of the model over the training iteration on the BPI-Challenge Dataset from 2019

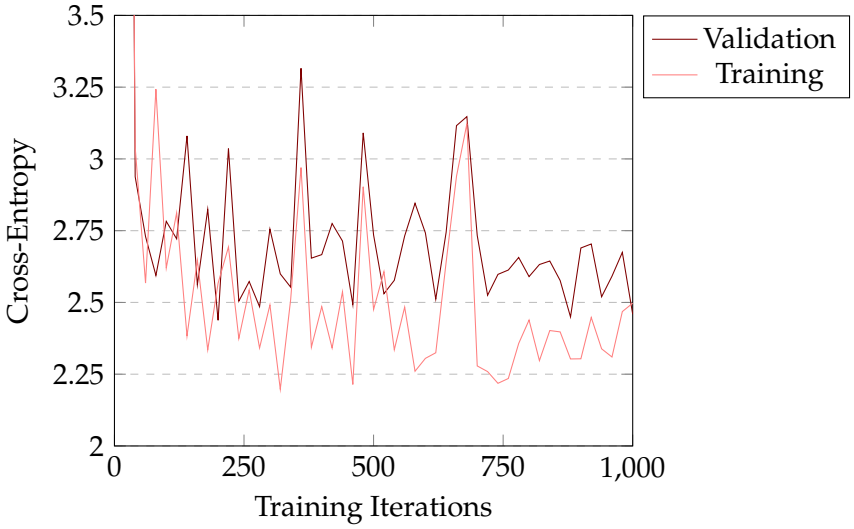


Figure 5.6: Cross-Entropy of the model over the training iteration on the BPI-Challenge Dataset from 2019

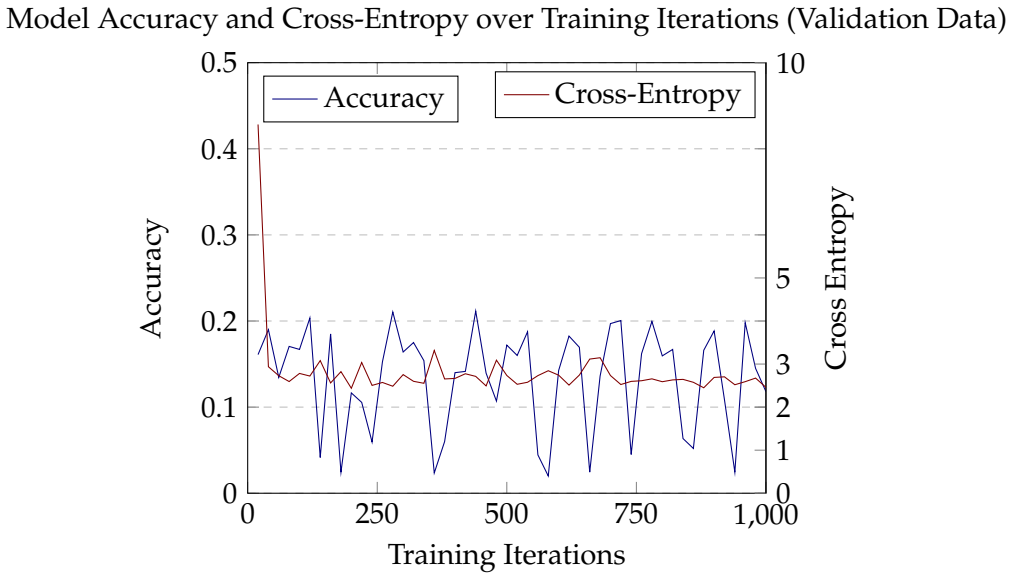


Figure 5.7: Performance of the model over the training iteration on the BPI-Challenge Dataset from 2019

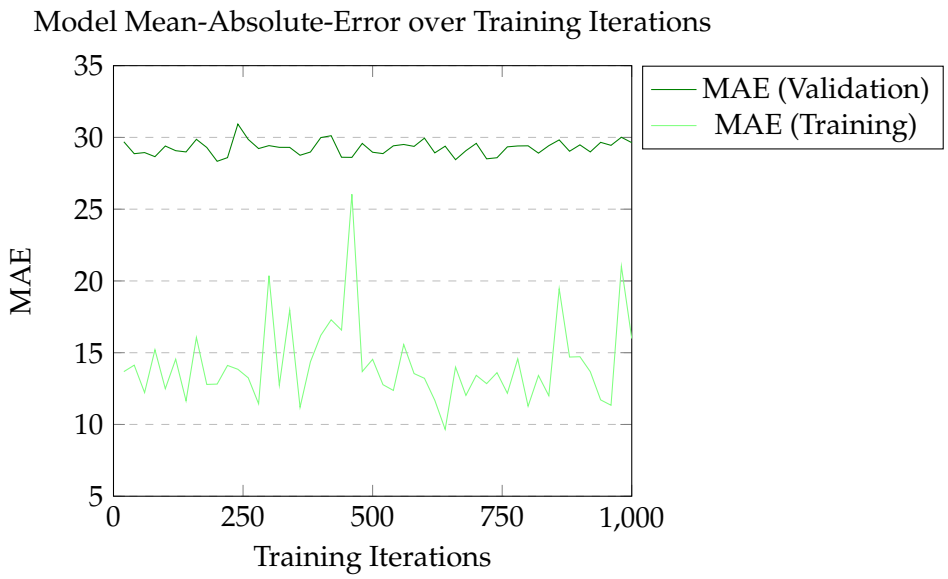


Figure 5.8: Performance of the model over the training iteration on the BPI-Challenge Dataset from 2019

A notable observation in both training instances is the substantial difference between MAE values for the training and validation datasets. This discrepancy suggests that the model, while potentially fitting well to the training data, encounters challenges in generalising its predictions to unseen instances. Unlike accuracy and cross-entropy, MAE values display a stability throughout the training process. This stability is evident from the consistent nature of the MAE curve, maintaining an approximately constant level with slight fluctuations. The consistently higher MAE values for the validation dataset indicate that the model's predictions for event and trace properties diverge more from the ground truth in the validation set compared to the training set. This divergence could stem from the model's limited exposure to diverse patterns during training. The stability in MAE values implies that, while the model maintains a consistent level of prediction error, this error remains relatively unchanged across training iterations.

5.4 Interpretation

The evaluation results shed light on critical aspects of the model's performance and its suitability for the intended task. The model's architecture appears to pose challenges in effectively capturing and predicting the intricacies of event and trace properties within the given task. The complexities of business processes may demand a more sophisticated and nuanced model architecture. The consistent fluctuation in accuracy, cross-entropy, and MAE values across training iterations suggests that while hyperparameter optimisation may contribute to performance improvements, fundamental adaptations to the model's architecture might be necessary. The observed limitations in the model's ability to generalise to unseen data, as indicated by the notable difference between training and validation performance, point towards potential issues in the current design's capacity to grasp diverse patterns present in the data. The evaluation results underscore the necessity to iterative development of the model and possible adaption to its architecture.

The observed limitations in the model's performance could be attributed to various factors, and exploring potential reasons provides insights into areas for improvement:

- **Information Compression and Loss:**
  The current tokenisation approach, where properties related to events or traces are transformed into a single token, might introduce information compression and loss. Embedding properties directly instead of transforming them into tokens could alleviate this issue. By maintaining the original dimensionality of property vectors, the model may better capture nuanced relationships.

- **Mixed Nature of Prediction Tasks:** The model's underlying tasks involve a combination of regression and classification. Predicting identifiers is essentially a classification task, while predicting trace and event properties involves regression. Separating these tasks or designing a model capable of handling both effectively is crucial. An alternative approach could involve dividing the model into multiple interconnected parts, each specialised in a specific prediction task.

- **Property Token Evaluation:**
  Evaluating property tokens as regression problems, even for attributes represented as strings or ids, might not be optimal. Incorporating lookup tables introduces classification elements, and adapting the loss function to handle both regression and classification aspects could be challenging. A potential solution involves modifying the model's architecture to treat properties as separate tokens, allowing for a clear distinction between regression and classification tasks.
- **Complexity and Training Challenges:**
  Balancing model complexity and training efficiency is critical. Introducing more intricate architectures or loss functions might improve performance but could also complicate the training process. Finding a balance between model complexity and training simplicity is essential to ensure efficient learning.
- **Alternative Architectures:**
  Exploring alternative model architectures tailored to the specific nature of business processes could be fruitful. While the Attention mechanisms is relativly new, and posses as a possible base architecture, other new architecture such as Mamba **??**, which embraces selective state spaces, could be used aswell.
- **Data Characteristics:**
  Examining the characteristics of the input data is essential. The comprehension of the distribution of events, traces, and their characteristics, as well as potential outliers or anomalies, and their potential changing characteristics over time, could lead to the development of a more robust model. This is because the division of the data into validation and training samples could effectively disseminate the characteristics of the entire data set onto the splits.

In summary, the suboptimal results may stem from a combination of information compression, mixed prediction tasks, challenges in property token evaluation, and the need for alternative

architectures. Addressing these issues through modifications in tokenisation, task separation, loss function adaptation, and exploring alternative architectures could pave the way for a more effective model in PPM.

6 Conclusion

The exploration of PPM within the broader context of PM has unveiled both the potential and challenges associated with leveraging ML and DL techniques. As the field of DL evolves, new opportunities arise for enhancing PPM approaches, aligning their capabilities with the advancements in the DL sector. The thesis explored the attention mechanism and transformers, highlighting their significance in the context of PPM. Multiple approaches, using the attention mechanisms for event prediction, were examined, demonstrating the versatility and applicability of these mechanisms in capturing intricate relationships within event logs. The central proposal of the thesis introduced a transformer-based architecture designed to harness the attention mechanism for sequence prediction. By abstracting the entire event log into token sequences, the model aimed to treat event identifiers, event properties, and trace properties as distinct tokens, transforming the predictive task into a comprehensive sequence prediction task. The rationale behind this abstraction drew inspiration from the success of similar approaches in NLP tasks, where tokenisation facilitates the understanding of relationships and differences between words.

Nonetheless, the assessment of the proposed model indicated that the present architecture may not be optimally suited for the task of PPM. Several potential reasons for the suboptimal performance were identified, including information compression of properties and the mixed nature of prediction tasks involving both regression and classification. These challenges pave the way for future work, offering opportunities for refinement and optimisation. The future development of the proposed architecture may entail the modification of the tokeniser's architecture and the projection of outputs, as the fundamental steps involved in tokenisation and de-tokenisation play a pivotal role in determining the model's comprehension and prediction capabilities. By addressing the identified obstacles and exploring alternative approaches, a model may attain a more accurate representation of event logs. Despite the setbacks in model performance, the exploration of PPM in this thesis could serve as a starting point for leveraging the notion of a token-based "Process Language". The iterative nature of model refinement, guided by insights from both successes and challenges, forms the basis for continuous progress in enhancing predictive capabilities. The advancements in DL architectures, approaches, and their applications promise to be a catalyst for further research and appliances in the domain of PPM, significantly shaping the landscape of PPM.

List of Tables

List of Figures

Bibliography

[1] F. W. Taylor, *The principles of scientific management*. NuVision Publications, LLC, 1911. [Online]. Available: https://archive.org/details/principlesofscie00taylrich/mode/2up?view=theater

[2] A. Smith, *The theory of moral sentiments*. Penguin, 2010.

[3] ——, *The wealth of nations [1776]*. na, 1937, vol. 11937.

[4] P. B. Petersen, "Total quality management and the deming approach to quality management," *Journal of management History*, vol. 5, no. 8, pp. 468–488, 1999.

[5] H. J. Johansson, "Business process reengineering: Breakpoint strategies for market dominance," *(No Title)*, 1993.

[6] J. P. Womack, D. T. Jones, and D. Roos, *The machine that changed the world: The story of lean production–Toyota's secret weapon in the global car wars that is now revolutionizing world industry*. Simon and Schuster, 2007.

[7] J. P. Womack and D. T. Jones, "Lean thinking—banish waste and create wealth in your corporation," *Journal of the Operational Research Society*, vol. 48, no. 11, pp. 1148–1148, 1997.

[8] H. R. Max Roser and E. Mathieu. (2023, 03) What is moore's law? exponential growth is at the heart of the rapid increase of computing capabilities. [Online]. Available: https://ourworldindata.org/moores-law

[9] C. Walter, "Kryder's law," *Scientific American*, vol. 293, no. 2, pp. 32–33, 2005.

[10] M. Weske, *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.

[11] J. M. H. A. R. Marlon Dumas, Marcello La Rosa, *Fundamentals of Business Process Management*. Springer, 2013, 2018.

[12] *Process Mining Handbook*. Springer, 2022.

[13] W. M. van der Aalst, "Process mining: a 360 degree overview," in *Process Mining Handbook*. Springer, 2022, pp. 3–34.

[14] P. Zerbino, A. Stefanini, and D. Aloini, "Process science in action: A literature review on process mining in business management," *Technological Forecasting and Social Change*, vol. 172, p. 121021, 2021.

[15] W. M. van der Aalst, "Foundations of process discovery," in *Process Mining Handbook*. Springer, 2022, pp. 37–75.

[16] J. Carmona, B. van Dongen, and M. Weidlich, "Conformance checking: foundations, milestones and challenges," in *Process Mining Handbook*. Springer, 2022, pp. 155–190.

[17] V. Venkatesh, "Adoption and use of ai tools: a research agenda grounded in utaut," *Annals of Operations Research*, pp. 1–12, 2022.

[18] A. Zuiderwijk, Y.-C. Chen, and F. Salem, "Implications of the use of artificial intelligence in public governance: A systematic literature review and a research agenda," *Government Information Quarterly*, vol. 38, no. 3, p. 101577, 2021.

[19] I. J. B. Young, S. Luz, and N. Lone, "A systematic review of natural language processing for classification tasks in the field of incident reporting and adverse event analysis," *International journal of medical informatics*, vol. 132, p. 103971, 2019.

[20] N. C. Dang, M. N. Moreno-García, and F. De la Prieta, "Sentiment analysis based on deep learning: A comparative study," *Electronics*, vol. 9, no. 3, p. 483, 2020.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[22] OpenAI. Openai's chatgpt. [Online]. Available: https://chat.openai.com/

[23] R. Agrawal, D. Gunopulos, and F. Leymann, "Mining process models from workflow logs," in *International Conference on Extending Database Technology*. Springer, 1998, pp. 467–483.

[24] W. Van der Aalst, "Process design by discovery: Harvesting workflow knowledge from ad-hoc executions," in *Knowledge Management: An Interdisciplinary Approach, Dagstuhl Seminar Report*, no. 281. Citeseer, 2000.

[25] P. Tools. Prom tools. [Online]. Available: https://promtools.org/

[26] I. T. F. on Process Mining. (2012) Process mining manifesto. [Online]. Available: https://www.tf-pm.org/resources/manifesto

[27] IEEE. (2016, 11) Ieee 1849-2016 xes standard. [Online]. Available: https://xes-standard.org/

[28] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[29] OpenAi. gpt-2 (codebase). [Online]. Available: https://github.com/openai/gpt-2

[30] A. Karpathy. nanogpt. [Online]. Available: https://github.com/karpathy/nanoGPT

[31] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[33] C. Di Francescomarino and C. Ghidini, *Predictive Process Monitoring*. Cham: Springer International Publishing, 2022, pp. 320–346. [Online]. Available: https://doi.org/10.1007/978-3-031-08848-3_10

[34] D. A. Neu, J. Lahann, and P. Fettke, "A systematic literature review on state-of-the-art deep learning methods for process prediction," *Artificial Intelligence Review*, pp. 1–27, 2022.

[35] N. Ogunbiyi and A. Basukoski, "A context-aware process mining predictive model."

[36] A. Augusto, J. Carmona, and E. Verbeek, "Advanced process discovery techniques," in *Process mining handbook*. Springer, 2022, pp. 76–107.

[37] M. Ceci, P. F. Lanotte, F. Fumarola, D. P. Cavallo, and D. Malerba, "Completion time and next activity prediction of processes using sequential pattern mining," in *Discovery Science: 17th International Conference, DS 2014, Bled, Slovenia, October 8-10, 2014. Proceedings 17*. Springer, 2014, pp. 49–61.

[38] A. Rogge-Solti and M. Weske, "Prediction of business process durations using non-markovian stochastic petri nets," *Information Systems*, vol. 54, pp. 1–14, 2015.

[39] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[40] H. Wang and B. Raj, "On the origin of deep learning," *arXiv preprint arXiv:1702.07800*, 2017.

[41] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[42] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. Psychology press, 2005.

[43] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[44] P. J. Werbos, "Generalization of backpropagation with application to a recurrent gas market model," *Neural networks*, vol. 1, no. 4, pp. 339–356, 1988.

[45] ——, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[46] V. Pasquadibisceglie, A. Appice, G. Castellano, and D. Malerba, "Predictive process mining meets computer vision," in *Business Process Management Forum: BPM Forum 2020, Seville, Spain, September 13–18, 2020, Proceedings 18*. Springer, 2020, pp. 176–192.

[47] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[48] A. Malinowski, T. J. Cholewo, and J. M. Zurada, "Capabilities and limitations of feedforward neural networks with multilevel neurons," in *1995 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 1. IEEE, 1995, pp. 131–134.

[49] W. Kratsch, J. Manderscheid, M. Röglinger, and J. Seyfried, "Machine learning in business process monitoring: a comparison of deep learning and classical approaches used for outcome prediction," *Business & Information Systems Engineering*, vol. 63, pp. 261–276, 2021.

[50] J. Ezpeleta, J. Fabra, and P. Álvarez, "On the use of log-based model checking, clustering and machine learning for process behavior prediction," in *2018 Fifth International Conference on Social Networks Analysis, Management and Security (SNAMS)*. IEEE, 2018, pp. 209–214.

[51] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[52] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions," *Journal of big Data*, vol. 8, pp. 1–74, 2021.

[53] A. Al-Jebrni, H. Cai, and L. Jiang, "Predicting the next process event using convolutional neural networks," in *2018 IEEE International Conference on Progress in Informatics and Computing (PIC)*.

IEEE, 2018, pp. 332–338.

[54] N. Di Mauro, A. Appice, and T. M. Basile, "Activity prediction of business process instances with inception cnn models," in *AI* IA 2019–Advances in Artificial Intelligence: XVIIIth International Conference of the Italian Association for Artificial Intelligence, Rende, Italy, November 19–22, 2019, Proceedings 18*. Springer, 2019, pp. 348–361.

[55] A. Sherstinsky, "Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, 2020.

[56] A. Metzger, A. Neubauer, P. Bohn, and K. Pohl, "Proactive process adaptation using deep learning ensembles," in *Advanced Information Systems Engineering: 31st International Conference, CAiSE 2019, Rome, Italy, June 3–7, 2019, Proceedings 31*. Springer, 2019, pp. 547–562.

[57] C. Di Francescomarino, C. Ghidini, F. M. Maggi, G. Petrucci, and A. Yeshchenko, "An eye into the future: leveraging a-priori knowledge in predictive business process monitoring," in *Business Process Management: 15th International Conference, BPM 2017, Barcelona, Spain, September 10–15, 2017, Proceedings 15*. Springer, 2017, pp. 252–268.

[58] W. Merrill and A. Sabharwal, "The parallelism tradeoff: Limitations of log-precision transformers," *Transactions of the Association for Computational Linguistics*, vol. 11, pp. 531–545, 2023.

[59] S. Jamil, M. Jalil Piran, and O.-J. Kwon, "A comprehensive survey of transformers for computer vision," *Drones*, vol. 7, no. 5, p. 287, 2023.

[60] S. Latif, A. Zaidi, H. Cuayahuitl, F. Shamshad, M. Shoukat, and J. Qadir, "Transformers in speech processing: A survey," *arXiv preprint arXiv:2303.11607*, 2023.

[61] P.-X. Cai, Y.-C. Fan, and F.-Y. Leu, "Compare encoder-decoder, encoder-only, and decoder-only architectures for text generation on low-resource datasets," in *Advances on Broad-Band Wireless Computing, Communication and Applications: Proceedings of the 16th International Conference on Broad-Band Wireless Computing, Communication and Applications (BWCCA-2021)*. Springer, 2022, pp. 216–225.

[62] Z. Fu, W. Lam, Q. Yu, A. M.-C. So, S. Hu, Z. Liu, and N. Collier, "Decoder-only or encoder-decoder? interpreting language model as a regularized encoder-decoder," *arXiv preprint arXiv:2304.04052*, 2023.

[63] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[64] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[65] Z. A. Bukhsh, A. Saeed, and R. M. Dijkman, "Processtransformer: Predictive business process monitoring with transformer network," *arXiv preprint arXiv:2104.00721*, 2021.

[66] G. Rivera Lazo and R. Ñanculef, "Multi-attribute transformers for sequence prediction in business process management," in *International Conference on Discovery Science*. Springer, 2022, pp. 184–194.

[67] W. Ni, G. Zhao, T. Liu, Q. Zeng, and X. Xu, "Predictive business process monitoring approach based on hierarchical transformer," *Electronics*, vol. 12, no. 6, p. 1273, 2023.

[68] J. Wang, C. Lu, B. Cao, and J. Fan, "Mitfm: A multi-view information fusion method based on

transformer for next activity prediction of business processes," in *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, 2023, pp. 281–291.

[69] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[70] O. AI, "Openai (2023) - gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[71] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[72] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[73] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Barnes, and A. Mian, "A comprehensive overview of large language models," *arXiv preprint arXiv:2307.06435*, 2023.

[74] K. S. Kalyan, A. Rajasekharan, and S. Sangeetha, "Ammus: A survey of transformer-based pretrained models in natural language processing," *arXiv preprint arXiv:2108.05542*, 2021.

[75] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," *arXiv preprint arXiv:1801.06146*, 2018.

[76] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[77] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 19–27.

[78] P. Rodríguez, M. A. Bautista, J. Gonzalez, and S. Escalera, "Beyond one-hot encoding: Lower dimensional target embedding," *Image and Vision Computing*, vol. 75, pp. 21–31, 2018.

[79] M. K. Dahouda and I. Joe, "A deep-learned embedding technique for categorical features encoding," *IEEE Access*, vol. 9, pp. 114 381–114 391, 2021.

[80] S. J. Mielke, Z. Alyafeai, E. Salesky, C. Raffel, M. Dey, M. Gallé, A. Raja, C. Si, W. Y. Lee, B. Sagot *et al.*, "Between words and characters: a brief history of open-vocabulary modeling and tokenization in nlp," *arXiv preprint arXiv:2112.10508*, 2021.

[81] A. Rai and S. Borah, "Study of various methods for tokenization," in *Applications of Internet of Things: Proceedings of ICCCIOT 2020*. Springer, 2021, pp. 193–200.

[82] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," *arXiv preprint arXiv:1808.06226*, 2018.

[83] OpenAI. tiktoken. [Online]. Available: https://github.com/openai/tiktoken

[84] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[85] R. Xiong, Y. Yang, D. He, K. Zheng, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T. Liu, "On layer normalization in the transformer architecture," in *International Conference on Machine Learning*. PMLR, 2020, pp. 10 524–10 533.

[86] S. Xie, H. Zhang, J. Guo, X. Tan, J. Bian, H. H. Awadalla, A. Menezes, T. Qin, and R. Yan, "Residual: Transformer with dual residual connections," *arXiv preprint arXiv:2304.14802*, 2023.

[87] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[88] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[89] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring massive multitask language understanding," *arXiv preprint arXiv:2009.03300*, 2020.

Appendices

Appendix 1

```
for($i=1; $i<123; $i++)
{
    echo "work harder! ;)";
}
```