POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

MACHINE LEARNING FOR ANOMALY DETECTION IN KERNEL TRACES

QUENTIN FOURNIER
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Décembre 2022

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

MACHINE LEARNING FOR ANOMALY DETECTION IN KERNEL TRACES

présentée par **Quentin FOURNIER**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

DEDICATION

To my parents
for their support
and sacrifices...

ACKNOWLEDGEMENTS

The accomplishment of this thesis would not have been possible without the support of incredible people whose help had directly or indirectly contributed to my Ph.D. I ought to acknowledge them and offer them my most sincere gratitude.

I would first like to express my most profound gratitude to my parents, Agnès and Christophe, and to my brother, Mathias, for their wonderful support and encouragement. This endeavor would not have been possible without you.

I would also like to express my most sincere appreciation to my research advisers, Professors Daniel Aloise and Michel R. Dagenais, for allowing me to pursue a Ph.D. and offering me numerous academic opportunities. I could not have undertaken this journey without your guidance and support.

I would like to extend my gratitude to all my friends, Arnaud, Camille, Erica, Irving, Leandro, Paul, Pierre-Frederick, Rodrigo, and Sebastien, for these years of friendship. I will always be grateful for our fascinating discussions and your dedicated support.

This thesis would not have been possible without the advice and hard work of all staff at the Dorsal Lab, Arnaud Fiorini, Geneviève Bastien, and Naser Ezzati Jivan, and at Ciena, Anurag Prakash, François Tetreault, and Vahid Seyed Azhari.

I would like to acknowledge Professor Christian Raymond for introducing me to the world of research, encouraging me to pursue a Ph.D., and trusting me by offering my first research internship.

Lastly, I would like to recognize Professors Amal Zouaq, Christian Raymond, and Christopher J. Pal for accepting the invitation to compose my thesis jury.

RÉSUMÉ

En raison de l'étendue actuelle des systèmes informatiques, il existe un réel besoin de détecter efficacement toute déviation de leur comportement attendu. En effet, les interactions complexes entre le matériel et les logiciels entraînent souvent des comportements nouveaux et inattendus, y compris des comportements courants tels que des mises à jour logicielles, de nouveaux utilisateurs et des requêtes rares, mais aussi des anomalies telles que de mauvaises configurations, des latences, des intrusions, et des bogues.

Le traçage est une approche légère et efficace pour enregistrer le comportement des systèmes informatiques au moment de l'exécution en collectant des événements de bas niveau générés chaque fois qu'une instruction spécifique appelée point de trace est exécutée. En particulier, cette thèse étudie un sous-ensemble d'événements générés par le noyau Linux appelés *appels système*. Les appels système comprennent un nom et plusieurs arguments tels qu'un horodatage et un identifiant de processus, et correspondent à des requêtes d'applications s'exécutant dans l'espace utilisateur vers le noyau afin d'accéder à des ressources telles que la mémoire ou le réseau. Le principal avantage des appels système est qu'ils exposent le comportement de l'ensemble du système sans avoir à modifier le code source, car le noyau Linux a déjà été instrumenté avec des points de trace. De nos jours, de nombreux chercheurs considèrent les appels système comme la source d'informations la plus détaillée et la plus précise pour analyser les systèmes informatiques.

La première contribution de cette thèse est une méthode pour extraire le comportement interne des requêtes Web et une chaîne de traitement pour détecter les problèmes de performances qui permet également d'identifier leur cause. Tout d'abord, des informations de bas niveau sont collectées en traçant les espaces utilisateur et noyau du serveur Web. Ensuite, le chemin critique de chaque requête est calculé à partir duquel des attributs de plus haut niveau appelés *états d'exécution* sont extraits. Les valeurs aberrantes sont détectées et regroupées sur la base de ces caractéristiques, et le comportement de chaque groupe de valeurs aberrantes est analysé séparément. Les expériences réalisées ont révélé que ce pipeline est capable de détecter les requêtes Web lentes et de fournir des informations sur leurs causes. Notamment, un véritable problème de conflit de cache PHP a été découvert en utilisant l'approche proposée.

Les attributs fabriqués à la main par des experts sont connus pour être sous-optimaux, comme en témoigne le succès des réseaux de neurones qui apprennent automatiquement la représentation des attributs adaptée pour le problème. Par conséquent, les chercheurs ont proposé des réseaux de neurones qui apprennent une représentation compacte des appels système appelée plongement lexical. Cependant, l'efficacité de ces représentations est limitée par l'omission des *arguments* des appels système. La raison principale est que la communauté n'a pas de représentation compacte et de dimension fixe des arguments des appels système qui est adaptée à l'apprentissage profond. En guise de solution, la deuxième contribution de cette thèse est une approche générale pour apprendre une représentation des noms des appels système avec leurs arguments en utilisant à la fois le plongement lexical et l'encodage. La méthode proposée est facilement applicable à la plupart des réseaux de neurones et est indépendante de la tâche. Les expériences ont montré que notre approche améliorait la performance jusqu'à 11,3% sur deux tâches de modélisation de langage non supervisées avec deux réseaux de neurones largement utilisés, le LSTM et le Transformer.

Enfin, la troisième contribution de cette thèse étend la seconde en introduisant une méthodologie de détection de nouveauté qui repose sur une distribution de probabilité des séquences d'appels système, que l'on peut interpréter comme un modèle de langage. Les modèles de langage permettent d'estimer la probabilité des séquences, et puisque les nouveautés divergent des comportements précédemment observés par définition, elles sont peu probables d'après le modèle. Suivant la majorité de la littérature, la méthodologie proposée a été évaluée avec un LSTM. Étant donné que les traces du noyau sont généralement beaucoup plus longues que le contexte des LSTM, la méthodologie proposée a aussi été évaluée avec un Transformer qui est capable de modéliser des dépendances de longueur arbitraire au prix d'une complexité quadratique par rapport à la longueur de la séquence. Pour alléger la charge de calcul, un Transformer de complexité linéaire appelé Longformer a été choisi en fonction des motifs d'attention appris par le Transformer. Les réseaux de neurones nécessitent généralement une grande quantité de données pour être entraînés efficacement, et à notre connaissance, aucun ensemble de données moderne et massif de traces du noyau n'a été rendu public. Afin de remédier à cette limitation, un nouvel ensemble de données de traces du noyau comprenant plus de 2 millions de requêtes Web avec sept comportements distincts a été collecté et rendu disponible. La méthodologie proposée atteint un F-score et une AuROC supérieurs à 95% sur la plupart des nouveautés tout en étant indépendante du type de donnée et de nouveauté, en plus de nécessiter le moins possible un expert.

ABSTRACT

Due to the current extent of computer systems, there is a genuine need to effectively and efficiently detect deviations from their expected behavior. Indeed, complex interactions between hardware and software often result in novel and unexpected behaviors, including common behaviors such as software updates, new users, and rare queries, as well as anomalies such as misconfigurations, latency, intrusions, and bugs.

Tracing is a lightweight approach to recording the behavior of computer systems at runtime by collecting low-level events generated whenever a specific instruction called tracepoint is executed. This research focuses mainly on a subset of events generated by the Linux kernel called *system calls*. System calls comprise a name and multiple arguments, such as a timestamp and a process id, and correspond to requests from applications running in the userspace to the kernel in order to access resources such as memory or network. The main benefit of system calls is that they expose the behavior of the whole system without having to modify the source code, as the Linux kernel has already been instrumented with tracepoints. Nowadays, many researchers consider system calls to be the most fine-grained and accurate source of information to investigate computer systems.

The first contribution of this thesis introduces a method to extract the internal behavior of web requests and a pipeline to detect performance issues that also provide insights into their root cause. First, low-level and fine-grained information is collected by tracing the user and kernel spaces of the web server. Then, the critical path of each request is computed from which higher-level features called *execution states* are extracted. Outliers are detected and clustered based on these handcrafted features, and the behavior of each group of outliers is analyzed separately. Experiments revealed that this pipeline is able to detect slow web requests and provide additional insights into their true root causes. Notably, a real PHP cache contention issue was discovered using our proposed approach.

Features handcrafted by experts are known to be suboptimal, as demonstrated by the tremendous success of neural networks, which automatically learn the representation of the features for the task at hand. Accordingly, researchers have proposed neural networks that learn dense representations of system call names called embeddings. However, the effectiveness of these representations has been hindered by omitting the system calls *arguments*. The main reason is that the community does not have an appropriate compact and fixed-dimensional representation of the system call arguments. As a solution, the second contribution of this thesis introduces a general approach to learning a representation of the event names along with their arguments using both embedding and encoding. The proposed method is readily applicable to most neural networks and is task-agnostic. Experiments showed that our approach improved the performance by up to 11.3% on two unsupervised language modeling tasks with two widely-used neural networks, namely the LSTM and the Transformer.

Finally, the third contribution of this thesis extends over the second by introducing a novelty detection methodology that relies on a probability distribution over sequences of system calls, which one may interpret as a language model. Language models allow estimating the likelihood of sequences, and since novelties deviate from previously observed behaviors by definition, they are unlikely under the model. Following most of the literature, the proposed methodology was evaluated on an LSTM. Since kernel traces are typically much longer than the effective context length of LSTMs, the proposed methodology was also evaluated on a Transformer, which is able to model arbitrary-length dependencies at the cost of a quadratic complexity with respect to the sequence length. In order to mitigate this computational burden, a lower-complexity Transformer called the Longformer was chosen based on the attention patterns learned by the Transformer. Large neural networks typically require a massive amount of data to be trained effectively, and to the best of our knowledge, no modern and massive datasets of kernel traces are publicly available. In order to address this limitation, a new open-source dataset of kernel traces comprising over 2 million web requests with seven distinct behaviors is introduced. The proposed methodology achieves an F-score and AuROC greater than 95% on most novelties while being data- and task-agnostic, besides requiring minimal expert handcrafting.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

ACT	Adaptive Computation Time
AI	Artificial Intelligence
AIOps	Artificial Intelligence for IT Operations
ASR	Automatic Speech Recognition
BLEU	Bilingual Evaluation Understudy
BoW	Bag-of-Word
BPTT	Backpropagation Through Time
CCT	Context Calling Tree
CO2	Carbone Dioxide
CPU	Central Processing Unit
DACS	Decoder-end Adaptive Computation Steps
DAM	Differentiable Attention Mask
DAT	Depth-Adaptive Transformer
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
DL	Deep Learning
DNA	Deoxyribonucleic Acid
ECCT	Enhanced Context Calling Tree
ENAS	Efficient Neural Architecture Search
ETC	Extended Transformer Construction
FFN	Feed-Forward Network
FIFO	First-In First-Out
FLOP	Floating Point Operations
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
IO	Input Output
ISOMAP	Isometric Feature Mapping
ITC	Internal Transformer Construction
LM	Language Model
LSH	Locality-Sensitive Hashing
LSTM	Long Short-Term Memory
LTTng	Linux Trace Toolkit: next generation
MC	Matthews Correlation
MIPS	Maximum Inner Product Search

MLM	Masked Language Model
MLP	Multilayer Perceptron
ML	Machine Learning
MoE	Mixture-of-Experts
NAS	Neural Architecture Search
NLP	Natural Language Processing
NNS	Nearest Neighbour Search
PHP	PHP: Hypertext Preprocessor
RECL	Relative Effective Context Length
RNN	Recurrent Neural Network
SQL	Structured Query Language
ST	Speech Translation
TF-IDF	Term Frequency–Inverse Document Frequency
TIM	Transformers with Independent Mechanisms
TPU	Tensor Processing Unit
TTS	Text-to-Speech
WER	Word Error Rate

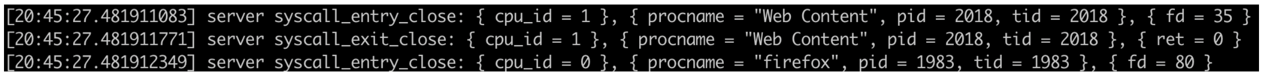
LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

Since the introduction of modern computers by Alan Turing in 1936, computer systems have become ever more complex. Even though they remain virtually deterministic, intricate interactions between hardware and software often result in *novel behaviors*. This research defines novel behaviors to be any deviation from previously observed behaviors. As such, novel behaviors include not only anomalies such as misconfigurations, latency, intrusions, hardware failures, and bugs but also common behaviors such as component upgrades, software updates, new users, and rare queries. Nowadays, computer systems are omnipresent in our society and essential in our daily lives, from medical diagnosis to banking systems. As a result, there is a genuine need to ensure the behavior of computer systems and thus to efficiently and effectively detect, classify, and explain novel behaviors.

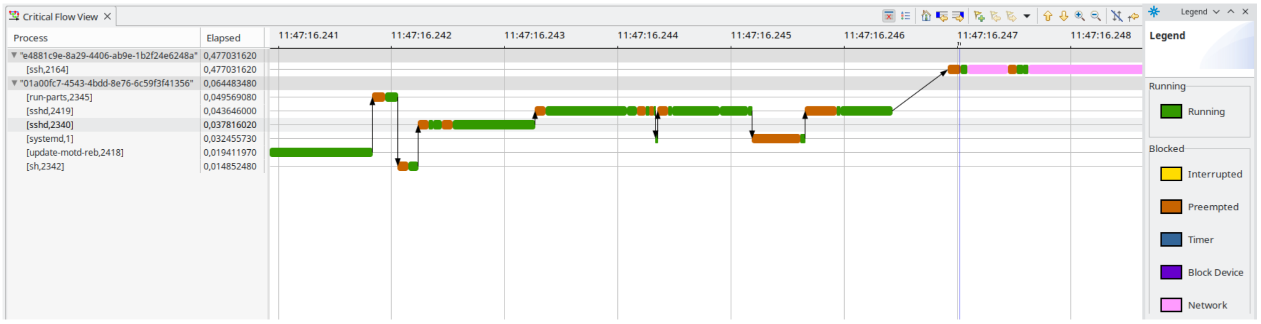
The behavior of computer programs is often investigated with debuggers and profilers. However, debuggers require stopping the program execution, which is not able to reveal issues related to latency, while profilers aggregate the collected metrics, which hides outliers. Instead, tracing is a lightweight approach to recording the behavior of computer systems at runtime by collecting low-level events generated whenever a specific instruction called tracepoint is executed. Such low-level events are either generated in the user space by applications or in the kernel space by the operating system. This research focuses on kernel events since they expose the system's behavior without requiring to manually instrument the source code, as most kernels already comprise tracepoints. In particular, this research relies on system calls that correspond to requests from applications running in the userspace to the kernel in order to access resources that would otherwise be inaccessible. As mentioned by Spinellis [1], the behavior of computer programs is mostly defined by its interactions with the operating system and errors can often be explained in terms of the system calls that were or were not issued. Nowadays, many researchers consider system calls to be the most fine-grained and accurate source of information to investigate computer systems.

Let us further describe the system calls and illustrate how they allow the detection of a resource contention and a bug causing a latency. System calls are events generated by the operating system and comprise a name and possibly many arguments. Several arguments are common to all system calls, such as the process id, the thread id, and the timestamp, while most are specific to each system call. For instance, the system call `read` comprises a file descriptor `fd` and the number of bytes `count` to be read in addition to the process id `pid`, the thread id `tid`, and the timestamp `timestamp`. Figure 1.1 illustrates three system calls as displayed by Babeltrace[1]. Some novelties and anomalies are identifiable with simple statistics, such as the number of system calls. For instance, a resource contention caused by multiple threads trying to access a shared cache concurrently will generate numerous system calls `fcntl` related to file control operations. However, most anomalies require considering the arguments. For instance, a bug causing a latency will increase the elapsed time between the system calls inside the critical path, and system calls with the highest increase in elapsed time may indicate the root cause of the bug. The critical path, also referred to as the active path, corresponds to the longest path between the elements of a network. In the case of project management, the critical path corresponds to the longest sequence of tasks that must be achieved to complete a project. In the case of traces, the critical path corresponds to the longest path of dependencies between the threads and resources. Figure 1.2 illustrate a toy critical path using Trace Compass[2]. However, most real-world anomalies and novelties require a more complex analysis.



```
[20:45:27.481911083] server syscall_entry_close: { cpu_id = 1 }, { procname = "Web Content", pid = 2018, tid = 2018 }, { fd = 35 }
[20:45:27.481911771] server syscall_exit_close: { cpu_id = 1 }, { procname = "Web Content", pid = 2018, tid = 2018 }, { ret = 0 }
[20:45:27.481912349] server syscall_entry_close: { cpu_id = 0 }, { procname = "firefox", pid = 1983, tid = 1983 }, { fd = 80 }
```

Figure 1.1 System calls as displayed by Babeltrace. The first two lines correspond to the start and end of the execution of the system call `close` requested by the process `Web Content`.



Process	Elapsed
▼ "e4881c9e-8a29-4406-ab9e-1b2f24e6248a"	0,477031620
[ssh,2164]	0,477031620
▼ "01a00fc7-4543-4bdd-8e76-6c59f3f41356"	0,064483480
[run-parts,2345]	0,049569080
[sshd,2419]	0,043646000
[sshd,2340]	0,037816020
[systemd,1]	0,032455730
[update-motd-reb,2418]	0,019411970
[sh,2342]	0,014852480

Figure 1.2 Critical path as displayed by Trace Compass.

Kernel traces are a formidable source of low-level information, and their manual inspection with tools such as Trace Compass may reveal insights about the system's behavior that would otherwise be inaccessible. However, their manual analysis is tedious and often prohibitive in practice. Indeed, due to the computational speed of modern computers, operating systems often execute hundreds of system calls every second. As a result, practitioners and researchers often must analyze traces automatically.

[1]`https://babeltrace.org`
[2]`https://www.eclipse.org/tracecompass/`

In addition to the high troughput of modern operating systems, hardware and software frequently evolve. For instance, a new CPU architecture is released every two to three years, and the Linux kernel is updated every two months. Therefore, detecting unexpected or unknown behaviors in such ever-changing environments is challenging to specify manually. Instead, machine learning techniques that learn how to solve a task from examples are better suited to analyze traces. Accordingly, this thesis investigates effective and efficient machine learning approaches to detect deviations from the expected behavior with kernel traces.

1.1 Research Objectives

Computer systems handle many critical applications, thus ensuring their behavior has become necessary. For instance, autonomous cars require the software and hardware to behave precisely and consistently as expected, or the consequences may be tragic. Therefore, numerous approaches have been explored to automatically detect and explore deviations in the behaviors of computer systems, most prominently unsupervised machine learning methods to detect anomalies. Nonetheless, detecting anomalies or novelties with kernel traces remains an open problem due to the size and complexity of the data. Accordingly, the three objectives of this research aim to investigate interesting novel machine learning approaches to detect anomalies and novelties. More specifically, this research is organized around three contributions:

1. **Anomaly detection with high-level features and off-the-shelf unsupervised methods.** In order to become familiar with the data and task at hand, the methodology of the first contribution closely follows the literature by introducing an approach to detect anomalies in web requests with high-level features and off-the-shelf unsupervised methods. Specifically, the execution states of each thread that contributes to the response time of each request are extracted from the critical path. These high-level features represent the internal behavior of the web requests and include *running*, *interrupt handling*, *waiting for disk*, *waiting for network*, *waiting for a timer*, and *waiting for another task*. The features are visualized with isometric mapping (ISOMAP), which already highlights some outliers. Density-based spatial clustering of applications with noise (DBSCAN) is applied to the handcrafted features to automatically detect outliers. The outliers are subsequently grouped into three clusters with k-means. Each group of deviations is analyzed separately with n-grams and simple metrics on the requests, allowing for the explanation of two out of the three groups of anomalies. The methodology is efficient as it was able to process about 50,000 requests in less than a minute while requiring minimal implementation since it relies on off-the-shelf methods.

2. **General approach to learning a representation of the event names along with their arguments.** Although the first contribution is efficient and effective in detecting latency, the methodology relies on handcrafted features which are often specific to the data or task, time-consuming, error-prone, and suboptimal. Instead of relying on handcrafted features, neural networks learn to extract relevant features for the task, thereby reducing the need for an expert and improving the model performance in most cases. Therefore, researchers have leveraged neural networks to learn an embedding of system call names. However, the effectiveness of the representation has been limited by discarding the arguments, arguably a valuable source of information. As a solution, the second contribution introduces a method for learning a representation of the event names along with their arguments that is readily applicable to most neural networks and is task-agnostic. Specifically, the proposed representation learns an embedding of the inherently meaningful arguments, such as the process name, and encodes the arguments whose meaning depends on the context, such as the process id. The impact of the arguments is quantified by means of an ablation study conducted on two datasets with two widely-used neural networks (LSTM and Transformer) and two language modeling tasks (left-to-right and masked LM). Experiments showed that the arguments improved the performance by up to 11.3%.

3. **Language models for novelty detection in kernel traces.** As revealed by the second contribution, learning a joint representation of the event names along with their arguments improves the effectiveness of neural networks, at least on language modeling tasks. A language model is a probability distribution over sequences of tokens, often words or characters, that allows estimating the likelihood of sequences. Since novelties deviate from previously observed behaviors by definition, they would have a low likelihood under a language model trained to maximize the likelihood of known behaviors. Therefore, the third contribution introduces a novelty detection methodology that relies on a probability distribution over sequences of system calls embedded with the representation technique introduced by the second contribution. The methodology is evaluated with three neural networks. Following the majority of the recent literature, the first network is based on the LSTM architecture. Since kernel traces are typically much longer than the effective context length of LSTMs, potentially limiting their effectiveness, a second model capable of modeling arbitrary-length dependencies called the Transformer is considered. The major benefit of Transformers comes at the cost of a quadratic complexity with respect to the sequence length. As a solution to the computational cost of the Transformer, a lower-complexity alternative called the Longformer has been selected based on the patterns learned by the Transformer. State-

of-the-art models, especially for natural language processing (NLP), have grown ever larger in terms of parameters. However, large neural networks typically require a considerable amount of data to be trained effectively, and to the best of our knowledge, no modern and massive datasets of kernel traces are publicly available. This limitation is addressed with a new open-source dataset of kernel traces comprising over 2 million web requests with seven distinct behaviors. The three neural networks achieve an F-score and AuROC greater than 95% on most novelties and are able to detect small latencies.

1.2 Thesis Outline

The remainder of this manuscript is organized as follows. Chapter 2 provides a critical review of the data collection and representation techniques, the machine learning models, and the actual anomaly or novelty detection schemes. Chapter 3 extends the literature review by extensively investigating techniques to improve the efficiency of state-of-the-art neural networks for sequence processing. Chapter 4 details the organization of this thesis. Chapter 5 introduces a simple anomaly detection approach with high-level features and off-the-shelf unsupervised methods. Chapter 6 introduces a methodology to learn a joint representation of event names and their arguments. Chapter 7 introduces a methodology to detect novelties with a language model that exploits the aforementioned joint representation. Chapter 8 provides a summary of the works and discusses the limitations and threats to validity. Finally, Chapter 9 concludes this thesis and proposes future research directions.

1.3 Publications

Chapters 3, 5, 6, and 7 of this thesis include the following articles, respectively.

- Quentin Fournier, Gaétan Marceau Caron and Daniel Aloise, "A Practical Survey on Faster and Lighter Transformers," ACM Computing Surveys, 2022, accepted with minor modifications.

- Quentin Fournier, Naser Ezzati-jivan, Daniel Aloise and Michel R. Dagenais, "Automatic Cause Detection of Performance Problems in Web Applications," 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2019, pp. 398-405, doi: 10.1109/ISSREW.2019.00102.

- Quentin Fournier, Daniel Aloise, Seyed Vahid Azhari and François Tetreault, "On Improving Deep Learning Trace Analysis with System Call Arguments," 2021 IEEE/ACM

18th International Conference on Mining Software Repositories (MSR), 2021, pp. 120-130, doi: 10.1109/MSR52588.2021.00025.

- Quentin Fournier, Daniel Aloise and Leandro Costa, "Language Models for Novelty Detection in Kernel Traces," Transactions on Modeling and Performance Evaluation of Computing Systems, 2022, under review.

Furthermore, collaborations during this research produced the following articles.

- Naser Ezzati-Jivan, Quentin Fournier, Michel R. Dagenais and Abdelwahab Hamou-Lhadj, "DepGraph: Localizing Performance Bottlenecks in Multi-Core Applications Using Waiting Dependency Graphs and Software Tracing," 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2020, pp. 149-159, doi: 10.1109/SCAM51674.2020.00022.

- Sneh Patel, Brendan Park, Naser Ezzati-Jivan and Quentin Fournier, "Automated Cause Analysis of Latency Outliers Using System-Level Dependency Graphs," 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), 2021, pp. 422-433, doi: 10.1109/QRS54544.2021.00054.

- Hervé Kabamba, Quentin Fournier and Michel R. Dagenais, "NodeCompass: Performance Analysis of Event-driven Multi- layered Single-threaded Systems," Transactions on Modeling and Performance Evaluation of Computing Systems, 2022, under review.

- Pierre-Frédérick Denys, Quentin Fournier and Michel R. Dagenais, "Distributed Computation of the Critical Path from Execution Traces," Software: Practice and Experience, 2022, under review.

- Mohammad Khanahmadi, Alireza Shameli-Sendi, Masoume Jabbarifar, Quentin Fournier and Michel R. Dagenais, "Detection of Microservice-Based Software Anomalies Based on Opentracing in Cloud," Software: Practice and Experience, 2022, under review.

- Ehsan Khodayarseresht, Alireza Shameli-Sendi, Quentin Fournier and Michel R. Dagenais, "DeEnergy and Carbon-Aware Initial VM Placement in Geographically Distributed Cloud Data Centers," Sustainable Computing: Informatics and Systems, 2022, under review.

CHAPTER 2 LITERATURE REVIEW

The research reported in this manuscript belongs to the field of artificial intelligence for IT operations[1] (AIOps), which is the use of artificial intelligence approaches such as machine learning models and natural language processing techniques to automate IT workflows. As such, this research is at the crossroad between software engineering and machine learning and comprises three fundamental aspects: (1) the data collection and representation, (2) the machine learning model, and (3) the actual anomaly or novelty detection scheme. This chapter presents a critical review of these three aspects, while the next chapter discusses the shortcomings of recurrent neural networks, describes the architecture of the Transformer, and extensively studies techniques to improve its efficiency. Although a comprehensive and critical literature review is provided in Chapters 2 and 3, please note that this manuscript is composed of self-contained chapters that include specific and detailed literature reviews.

Before proceeding further, let us preface the remainder of this section by discussing the distinction between anomaly and novelty. As defined in Chapter 1, novelties correspond to any deviation from previously observed behaviors and include common behaviors such as component upgrades, software updates, new users, and rare queries, as well as anomalies such as misconfigurations, latency, intrusions, hardware failures, and bugs. Even though novelties include anomalies, and their detection is therefore a broader problem, the literature has prominently focused on anomalies. As of the writing of this manuscript, one of the most popular approaches to detecting anomalies is to learn a "normal" behavior from the data and identify any deviations from this behavior as abnormal [2, 3, 4, 5, 6]. In our opinion, such approaches would be better framed as novelty detection methods. Furthermore, an additional mechanism would be necessary to determine whether the novel behaviors correspond to anomalies. Nonetheless, please note that this section follows the same terminology as the papers surveyed in order to be consistent with the literature.

2.1 Data Collection and Representation

2.1.1 Profilers, Debuggers, and Tracers

Profilers, debuggers, and tracers are the most commonly used tools by developers to investigate anomalies and novelties. This section briefly introduces each approach along with its strengths and limitations.

[1] `https://www.ibm.com/cloud/learn/aiops`

Debuggers are built into most integrated development environments (IDEs) and allow verifying the correctness of a piece of code. As explained by Araki et al. [7], debugging is an iterative process of developing and testing hypotheses, such as the expected behavior of a program or the location of a bug. First, developers must manually insert specific instructions called breakpoints into the source code. Then, the debugger stops the execution of the software every time a breakpoint is encountered and exposes the current variables. Next, the execution is either stopped, continued step by step, or resumed until another breakpoint is encountered. Although debuggers provide great insights into the internal behavior of software, they require to stop the execution, which makes them inappropriate for real-time applications and most performance anomalies. Besides, debuggers impose a significant overhead that may cause undesired latency.

Profilers are usually the tool of choice to quantify the resource usage of a piece of code and identify bottlenecks. First, profilers collect metrics at runtime about the software, for instance, the number of function calls or their duration. Note that profilers may also collect metrics about the system and hardware, such as the CPU time and the number of stalls. Then, once the execution is over, profilers output a statistical summary of the collected metrics called a profile. Profiling is efficient, thereby allowing to continuously profile production systems with minimal perturbations [8]. Nonetheless, the aggregation of the metrics may hide novelties, including the ones only occurring when some conditions are met. Furthermore, the root cause of novelties is often hidden by the aggregation or outside the scope of profilers.

Tracers address these shortcomings by collecting low-level events generated whenever a specific instruction called tracepoint is encountered at runtime. Simply put, tracing corresponds to system-wide and low-level logging with an enormous throughput since modern operating systems typically generate thousands of events per second. Traces provide insights into the execution of a piece of code and have been extensively used to detect and investigate novelties and anomalies. Unlike debuggers and profilers, tracers do not stop the execution nor aggregate the events. These advantages come at the expense of larger files, and although they can manually be explored similarly to logs, they often have to be automatically analyzed. The reader is referred to the comprehensive survey of Gebai and Dagenais [9] for an analysis of the design, implementation, and overhead of the most common tracers. Notably, this research relies on a lightweight and efficient open-source tracer developed at Polytechnique Montréal called the Linux Trace Toolkit: next generation (LTTng) [10].

2.1.2 User and Kernel Spaces

As previously explained, trace events are generated when a specific instruction called tracepoint is encountered at runtime. Specifically, a tracepoint or instrumentation point is an instruction that provides a hook to a light function specified at runtime called a probe. A probe is called whenever the hooked tracepoint is encountered, provided that it is enabled, to perform a custom task that is either implemented by the tracer or the user [9, 11]. Depending on the location of the tracepoint, events are either generated from the user space or the kernel space.

User space events are generated by tracepoints that have been added to the source code of the application prior to compilation (static instrumentation) or that have been inserted into the binary executable after compilation (dynamic instrumentation) [12]. The former requires access to the source code, while the latter is significantly more complex to implement. In both cases, developers must add and maintain the user space tracepoints themselves. Furthermore, user space instrumentation is unable to expose the behavior of the operating system (OS), which is instrumental in detecting some novelties and identifying their root cause. Indeed, several actions cannot be performed in the user space, including extending the process memory, opening a socket, locking a file, and waking up a thread. Moreover, user space events often greatly differ from one application to another as user space events are ad hoc.

Kernel space events are generated by tracepoints located in the kernel of the operating system. In particular, the Linux kernel has already been instrumented with over a thousand distinct tracepoints [9]. The main advantages of kernel space events are: (1) software developers do not need to instrument their code, (2) events are generic and consistent, and (3) the whole system is exposed [1]. Furthermore, kernel events are necessary to detect performance anomalies whose cause is external to the application, such as networking issues, database design issues, resource contentions, and hardware failures. However, kernel events correspond to low-level actions or changes in the system that cannot easily be linked to application-level operations. Besides, their throughput is typically greater than user space events as they correspond to the lowest level of information.

Tracers collect events from a single space or both spaces. In this work, only kernel events are considered due to their readiness, generality, and visibility of the system. In particular, a subset of kernel events known as system calls will be put under the microscope. System calls, or syscalls for short, are the entry points into the kernel for software to request a service from the operating system. In other words, system calls correspond to requests from applications running in the userspace to the kernel in order to access resources such as memory, network, or other devices that would otherwise be inaccessible. Note that a system call usually generates two events, one at the start and one at the end of its execution. Nowadays, many researchers consider system calls to be the most fine-grained and accurate source of information to analyze computer systems [2]. Appendix A provides the system call sequence associated with the simple `echo "Hello World"` command.

2.1.3 Natural Language Perspective and Challenges

Kernel traces are discrete sequences of events traditionally modeled with classical techniques such as high-level hand-crafted features and density plots [13] or fixed length patterns and rule-based models [14, 15]. The first contribution of this thesis follows the traditional approach by modeling kernel traces with execution states and off-the-shelf unsupervised methods.

Nonetheless, kernel traces also closely resemble natural languages in that they comprise both syntax and semantics. For instance, a file or a socket cannot be written into before it was open (syntax), and the system call `open` has a clear inherent meaning (semantic), which may vary depending on the arguments (polysemy). Therefore, kernel traces may be interpreted as a language, albeit not a natural one. In that case, traces correspond to documents, requests or spans correspond to sentences, and individual events correspond to words. Such a linguistic interpretation of kernel traces poses two challenges compared to natural languages.

The first challenge arises from the high throughput of modern operating systems. Indeed, since hundreds of events may be generated each second, a simple web request may comprise thousands of events, whereas natural language sentences typically comprise less than a hundred words. As a result, kernel traces may comprise dependencies that span much farther than in natural languages, making them more complex and expensive to model. Furthermore, the vocabulary size of kernel traces is much greater than that of natural languages, provided that arguments are included. For instance, the Linux kernel only comprises about 300 system calls, but each may comprise several arguments with a wide range of values. Although kernel traces are much more complex than natural languages, they are well-defined, stable, and easily collected.

The second challenge arises from the parallel nature of most modern computer systems. Indeed, CPU cores may execute system calls concurrently. Thanks to the high precision of the timestamp, two events are highly unlikely to be generated with the exact same timestamp. Nonetheless, two consecutive events generated by two distinct cores may be swapped in the trace without changing the meaning or behavior of the sequence. One way to address such high variability is to collect significantly more samples in order to learn which permutations are impactful.

2.1.4 Trace Representation

Kernel traces typically contain millions of low-level events with several arguments. Due to the high volume and complexity of kernel traces, researchers and practitioners have traded compactness over fine-grained information by discarding most arguments, aggregating events across time, and extracting higher-level features.

As of the writing of this thesis, most novelty and anomaly detection methodologies consider the event names but ignore the arguments, such as the process name, the process id, and the return value. However, the arguments are valuable information that allows the model to make more informed and, ultimately, more accurate predictions. Most prominently, temporal information encapsulated in some of the arguments, such as the timestamp, is essential to detecting performance issues. Consequently, several researchers considered with great success temporal information such as the timestamp [16], the duration [3], and the response time [17, 18]. Alternatively, Ezeme et al. [19] compressed the values of the arguments by encoding the characters using ASCII values and considering the frequency distribution of these values for each argument. Nonetheless, the second contribution of this research revealed the benefit of considering the actual values of multiple system call arguments, at least for neural language models [16].

In addition to reducing the number of arguments, several novelty and anomaly detection methods aggregate the events across time, thereby trading ordering information for a more compact representation. The most common approach is called bag-of-words (BoW), also known as *system call counts vector* [20], *frequency counts of system call names* [21] and *bag of system calls* [22]. Bag-of-words is a representation that corresponds to the number of occurrences of each element of a vocabulary in a document. For instance, the bag-of-word representation of the sequence $\boldsymbol{w} = \{a, c, c, c, a, c\}$ given the vocabulary $\mathbb{V} = \{a, b, c\}$ is $\{2, 0, 4\}$. Nonetheless, the ordering information is critical to detecting some anomalies, such as intrusions. Consequently, researchers have introduced methodologies that preserve part of the ordering. The most common technique is called n-gram, which makes the Markov assumption that events only depend on the $n-1$ previous events instead of all previous ones, thereby preserving only a short and local ordering. In particular, n-gram has been extensively used for intrusion detection as they are more expressive [21, 23, 24, 25]. Alternatively, Dymshits et al. [20] aggregated system calls on a short span, resulting in a sequence of system call counts. Nevertheless, numerous approaches do not aggregate events across time in order to preserve the complete ordering [2, 4, 17, 26, 13]

Independently of the aggregation level, the features provided to the machine learning algorithm are either: (1) straightforward, such as a one-hot-encoding [26] or a bag-of-words [20, 21, 22] of the system call names; (2) learned from the data, such as an embedding of the system call names [2, 4]; or (3) hand-crafted by an expert, such as states of kernel modules [13] or execution states [27]. Although carefully hand-crafted higher-level features may deliver excellent performance, they discard the fine-grained information that makes traces valuable to investigate the behavior of computer systems. Moreover, features are typically hand-crafted for a specific task and often do not perform well on other tasks and, as explained by Goodfellow et al. [28], "*manually designing features for a complex task requires a great deal of human time and effort; it can take decades for an entire community of researchers.*" Instead of relying on hand-crafted features, learning a relevant input representation for the task from data reduce the need for an expert, thereby reducing the human labor and risk of error while improving the performance in most cases.

2.2 Machine Learning Model

Let us clarify the distinction between the concepts of artificial intelligence (AI), machine learning (ML), and deep learning (DL).

Artificial intelligence is the oldest and most general concept of the three. Founded in the early '50s by the research of Alan Turing [29], the Oxford Dictionary defines artificial intelligence as "*the theory and development of computer systems able to perform tasks normally requiring human intelligence [...]*".

Machine learning, coined by Arthur Samuel in the late '50s [30], is a subset of artificial intelligence corresponding to the methods that learn to solve a task from data. Tom Mitchell defines machine learning in the eponym book as: "*A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*" [31].

Deep learning is, in turn, a subset of machine learning restricted to multi-layer neural networks. Such models learn a hierarchy of increasingly more complex concepts, each defined through its relation to the simpler ones. Contrary to popular belief, deep learning is not a modern concept: while the term was first used in 1986 [32], the first deep neural network was published in 1965 by Ivakhnenko and Lapa [33].

Since the turn of the millenium, numerous machine learning techniques have been explored to detect anomalies and novelties in logs and traces, including rule-based algorithms [14, 15], naive Bayes [34, 35], decision trees [35, 36], hidden Markov models [24, 35], and support vector machines (SVM) [35]. Due to the tremendous success of deep learning, researchers have recently shifted toward a family of neural networks called recurrent neural networks (RNNs) [37]. The most popular recurrent neural network is the LSTM [38] due to its ability to learn dependencies across a large number of time steps. The LSTM has been extensively and successfully used across many fields, including to detect anomalies and novelties in logs and traces. As of the writing of this survey, the LSTM is by far the most popular neural network to analyze traces [2, 17, 18, 19, 20, 36]. Alternatively, Lv et al. [26] used a gated recurrent unit (GRU) [39], which behaves similarly to the LSTM while requiring fewer parameters.

Chapter 3 describes the limitations of recurrent neural networks and discusses solutions that have been published, including the alignment between the input and output sequences solved with the sequence-to-sequence framework [40], the information bottleneck of the fixed-size-context [41] solved with the inter-attention mechanism [42], and the limited parallelizability solved with the introduction of another architecture called Transformer [43]. Nonetheless, the Transformer addresses the aforementioned limitations of the recurrent neural networks at the cost of a quadratic complexity with respect to the sequence length. Consequently, the following chapter further extensively surveys the approaches proposed to improve the efficiency of the Transformer.

2.3 Anomaly and Novelty Detection Scheme

Real-world anomalies and novelties are typically unknown. Indeed, developers usually solve anomalies after identifying them, while novelties are unknown by definition. Besides, manually labeling them afterward would be time-consuming and error-prone. As a result, their detection belongs to the unsupervised setting. However, a few supervised methods have been explored. Notably, Asmitha and Vinod [34] evaluated several supervised machine learning models, including naive Bayes, AdaBoost, and random forest, on a small labeled dataset. Furthermore, synthetic anomalies and novelties may be generated by injecting faults into the system. Nevertheless, the vast majority of the literature follows the unsupervised paradigm and the remainder of this section discusses the unsupervised detection schemes that have been explored.

A wide range of unsupervised anomaly and novelty detection methods output an outlier score quantifying how abnormal, novel, or outlier a sample is. By convention, the outlier score increases with the outlierness [44]. However, several studies proposed methods to measure such scores as the likelihood of sequences, which instead decreases with the outlierness as novelties and anomalies are unlikely. Therefore, note that the scores discussed hereafter may not correspond stricto sensu to outlier scores.

Numerous outlier scores have been explored, including the number of rule violations [14, 15], the number of event mispredictions [36, 6], the ratio of misspredictions [5], the probability of kernel states [13], the sequence negative log-likelihood [2]. In addition to the event mispredictions, prediction error of temporal information such as the timestamp [3] and the response time [17, 4] have been considered.

Once the outlier score has been computed, the decision is often made with a simple threshold on the outlier score. The threshold is often empirically computed on a validation set to maximize a metric such as accuracy or precision. Instead, Ezeme et al. [19] proposed a fancier way to compute the threshold based on the Bienaymé-Chebyshev inequality.

2.4 Discussion

In the literature, a wide range of machine learning algorithms have been applied to a wide range of features to detect specific anomalies. For instance, Murtaza et al. [13] explored states of kernel modules and density plots to detect host-based intrusions, while Murtaza et al. [35] explored the likelihood of some kernel events and decision trees to detect bugs in applications. The first contribution of this thesis follows this classical approach by investigating a novel combination of hand-crafted features and off-the-shelf machine learning methods to detect performance anomalies in web requests.

Nonetheless, researchers have recently moved away from hand-crafted features and instead focused on deep learning methods that learn a representation of the data for the task. Even though some techniques have considered the duration [3] and the response time [17, 18], no methods considered the actual value of multiple arguments. As explained by Dymshits et al. [20], the main reason for this shortcoming is that the community does "*[...] not have a compact fixed-dimensional representation for system call arguments suitable for large-volume training and classification.*" The second contribution of this thesis addresses this shortcoming with a method to learn a joint representation of the system call names and their arguments. Following the vast majority of the literature, the representation was evaluated on the LSTM. However, this recurrent neural network has been replaced by the Transformer for many sequence processing tasks, including the detection of anomalies in logs [6]. Still, the Transformer has yet to be explored to detect novelties or anomalies in kernel traces. Therefore, the third contribution of this thesis also evaluates the representation with a Transformer.

Several researchers have explored language models to detect various anomalies in kernel traces. Most prominently, Kim et al. [2] learned a language model of the system call names with LSTMs to detect intrusions based on the negative log-likelihood. The third contribution of this thesis extends the scope to novelty detection and improves over existing approaches based on language models in three ways. Firstly, the quality and quantity of data have been drastically improved, which is instrumental to scaling the neural networks. Secondly, neural networks that are able to learn extremely long dependencies are investigated, which could be instrumental in modeling some novelties. Thirdly, the novelty detection scheme relies on a metric that takes into account the sequence length called the perplexity.

Finally, the Transformer is a resource-intensive method due to its complexity. Therefore, a plethora of lighter and faster alternatives have been proposed and are described in the following chapter. Such models would be necessary to deploy the Transformer on real-world traces that may comprise thousands of events. Consequently, the third contribution also explores one such efficient alternative.

CHAPTER 3 ARTICLE 1: A PRACTICAL SURVEY ON FASTER AND LIGHTER TRANSFORMERS

Authors Quentin Fournier, Gaétan Marceau Caron, and Daniel Aloise.

Accepted with minor modifications at ACM Computing Surveys (27 January 2022)

Abstract Recurrent neural networks are effective models to process sequences. However, they are unable to learn long-term dependencies because of their inherent sequential nature. As a solution, Vaswani et al. introduced the Transformer, a model solely based on the attention mechanism that is able to relate any two positions of the input sequence, hence modelling arbitrary long dependencies. The Transformer has improved the state-of-the-art across numerous sequence modelling tasks. However, its effectiveness comes at the expense of a quadratic computational and memory complexity with respect to the sequence length, hindering its adoption. Fortunately, the deep learning community has always been interested in improving the models' efficiency, leading to a plethora of solutions such as parameter sharing, pruning, mixed-precision, and knowledge distillation. Recently, researchers have directly addressed the Transformer's limitation by designing lower-complexity alternatives such as the Longformer, Reformer, Linformer, and Performer. However, due to the wide range of solutions, it has become challenging for researchers and practitioners to determine which methods to apply in practice in order to meet the desired trade-off between capacity, computation, and memory. This survey addresses this issue by investigating popular approaches to make Transformers faster and lighter and by providing a comprehensive explanation of the methods' strengths, limitations, and underlying assumptions.

Keywords Deep Learning, Efficient Transformer, Self-Attention, Survey

3.1 Introduction

Sequences arise naturally in a wide range of domains, notably in natural language, biology, and software executions. Rumelhart et al. [37] introduced a family of models called recurrent neural networks (RNNs) based on the idea of parameter sharing to process variable-length sequences. Given an input sequence $\boldsymbol{X}$ comprising n tokens $\boldsymbol{x}^{(i)}$ of dimension d, recurrent neural networks iteratively construct a sequence of hidden representations $\boldsymbol{h}^{(i)}$ and produce a sequence of outputs $\boldsymbol{y}^{(i)}$ as illustrated in Figure 3.1. Unfortunately, vanilla RNNs often suffer from vanishing or exploding gradients, which prevent them from learning long-term dependencies. Hochreiter and Schmidhuber [38] addressed this limitation with the now widely popular long short-term memory (LSTM) network, which circumvents the gradient issues with paths through time. Cho et al. [39] later improved over the LSTM with the simpler gated recurrent unit (GRU).

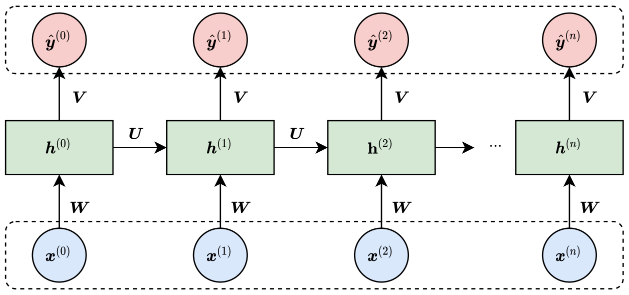


Figure 3.1 The computational graph of a recurrent neural network. The input and output sequences are depicted in blue and red, respectively. The position, also known as the time-step, is indicated in superscript. The weight matrices $\boldsymbol{W}$, $\boldsymbol{U}$, and $\boldsymbol{V}$ are shared across all positions. Reproduced with permission [16]. Copyright 2021 IEEE.

Recurrent neural networks align the input and output sequences, that is, there is a one-to-one mapping between the two sequences. Depending on the task, this property of RNNs may be too restrictive: for instance, translation requires outputting a sequence whose size is often different from that of the input while aligning tokens at different positions. Sutskever et al. [40] addressed this limitation by introducing the sequence-to-sequence framework in which a first network (encoder) processes the entire input sequence and returns its last hidden representation $\boldsymbol{h}^{(n)}$, effectively encoding the input into a fixed-size vector called context. The context then serves as the initial state for a second network (decoder), which generates the output sequence in an autoregressive manner. The decoding stops when a special end-of-sequence token is generated. Figure 3.2 illustrates the sequence-to-sequence framework.

In practice, the fixed-size nature of the hidden representation hinders the effectiveness of recurrent neural networks [41]. Indeed, as the input sequence is processed, information is iteratively stored into the hidden representation that may be too small to retain all the relevant information for the task. In that case, useful data is inevitably lost, which may significantly impact the model's performance. Bahdanau et al. [45] introduced an alignment mechanism called inter-attention to overcome the bottleneck of the sequence-to-sequence framework. This attention mechanism computes a different representation of the input for each output step, effectively allowing the decoder to "look at" the relevant part(s) of the input for each output step. Thereby, the inter-attention alleviates the encoder's burden to encode

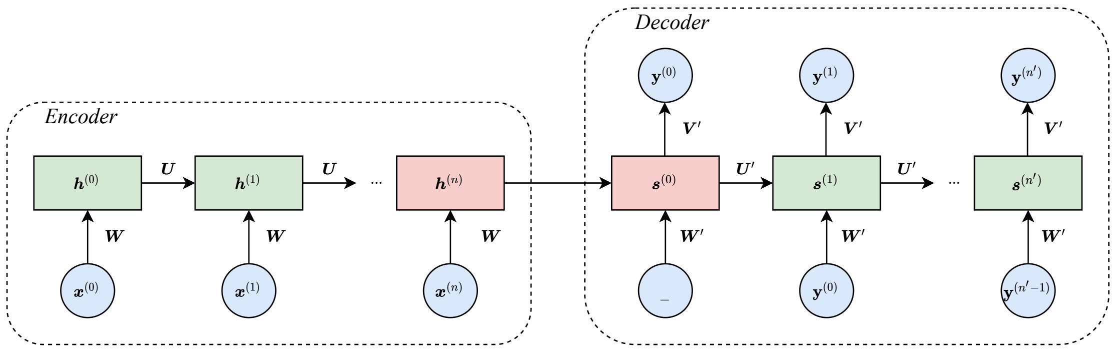


Figure 3.2 The sequence-to-sequence framework where the encoder and decoder are recurrent neural networks. The input sequence (blue) is encoded into a fixed-size context $\boldsymbol{h}^{(n)}$ (red), which serves as the initial state of the decoder.

all information about the input sequence into a fixed-size vector. Formally, the context is the weighted sum of the encoder's hidden representations $\boldsymbol{h}_i$, for $i = 1, \ldots, n$, where the weights are computed with a feed-forward neural network. For a comprehensive survey of the attention mechanism, we refer the reader to Galassi et al. [46] and Weng [47]. Figure 3.3 illustrates the inter-attention mechanism.

Moreover, recurrent neural networks do not scale efficiently to longer sequences due to their iterative nature [43]. In particular, RNNs struggle to learn dependencies between distant positions. One measure of this limitation is the relative effective context length (RECL) introduced by Dai et al. [48]. The RECL is the largest context length that leads to a substantial relative gain over the best model. In other words, increasing the context length over the RECL yields a negligible increase in performance over the best model. The authors estimated that the relative effective context length of LSTMs on natural language data is limited to approximately 400 words. Besides, Khandelwal et al. [49] empirically observed that LSTMs sharply model recent positions but only vaguely remember the distant past.

3.1.1 Transformer

This inherent limitation of recurrent neural networks has prevented them from being successfully applied to domains that require processing long sequences such as DNA. To overcome this limitation, Vaswani et al. [43] introduced the *Transformer*, a sequence-to-sequence model built without recurrences. Instead, the Transformer relies solely on the attention mechanism: the inter-attention between the encoder and decoder (see Figure 3.3), and the self-attention, also known as intra-attention, within the encoder and decoder. The self-attention's main advantage is its ability to relate any two positions of the input sequence regardless of their distance, thus increasing performance significantly on a wide range of tasks, including natural language processing (NLP) [50, 51, 43], computer vision [52, 53, 54], speech recognition [55, 56, 57], and biological sequence analysis [58]. Karita et al. [59] evaluated a Transformer against a sequence-to-sequence Bi-LSTM baseline on automatic speech recognition (ASR), speech translation (ST), and text-to-speech (TTS). The attention-based models outperformed the baseline on 13 corpora out of 15 for monolingual ASR and realized more than 10% relative improvement in 8 languages out of 10 for multilingual ASR. The Transformer improved the BLEU score from 16.5 for the baseline to 17.2 on ST while performing on par for TTS. Table 3.1 reports the performance improvements brought by popular Transformer architectures over previous state-of-the-art models across different domains. As of this paper's writing, the Transformer has become the de facto model for numerous sequence processing tasks.

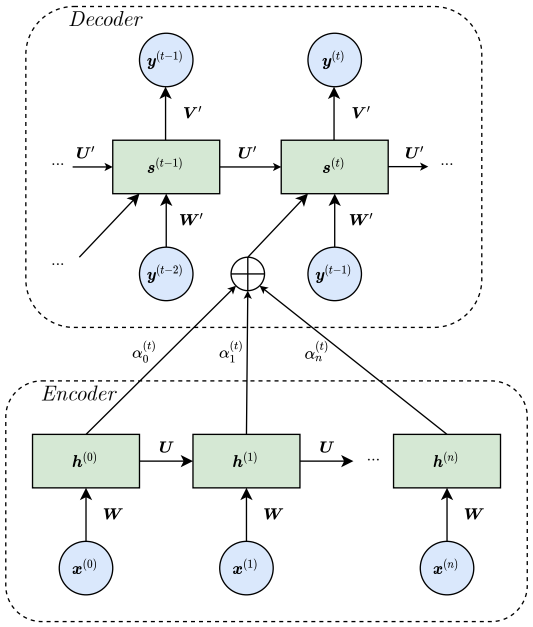


Figure 3.3 The inter-attention mechanism. The attention weight $\alpha_i^{(t)}$ depicts the strength with which the i-th encoder hidden representation $h^{(i)}$ contributes to the context of t-th decoder step. Reproduced with permission [16]. Copyright 2021 IEEE.

As an illustration of an end-to-end application of the Transformer, let us consider the speech recognition task. In hybrid approaches, the recognition system consists of independently trained machine learning components, often an acoustic model, a pronunciation model, and a language model. Instead, in end-to-end approaches, the recognition system consists of

[3]Bilingual evaluation understudy (BLEU), higher is better.
[4]Matthews correlation (MC) coefficient, higher is better.
[5]Word error rate (WER), lower is better.

Table 3.1 Relative improvements brought by popular Transformer architectures over previous state-of-the-art models. Absolute differences are reported between parenthesis. Sources are: [43] for machine translation, [53] for image classification, [51, 60] for text classification, and [61] for speech-to-text.

Task	Dataset	Previous SOTA	Transformer's Architecture	Relative Improvement
Machine Translation	newstest2014 (EN-to-DE)	MoE (GNMT) [62]	Vanilla [43]	9.1% (+2.37 BLEU[3])
	newstest2014 (EN-to-FR)			3.1% (+1.24 BLEU)
Image Classification	ImageNet	Noisy Student (EfficientNet-L2) [63]	ViT [53]	0.2% (+0.15% Acc)
	CIFAR-10	BiT-L (ResNet152x4) [64]		0.1% (+0.13% Acc)
	CIFAR-100			1.1% (+1.04% Acc)
	VTAB (19 tasks)			1.8% (+1.34% Acc)
Text Classification	SST2	Sparse byte mLSTM [65]	BERT[51]	1.8% (+1.70% Acc)
	CoLA	Single-task BiLSTM + ELMo + Attn [66]		72.9% (+25.5 MC[4])
Speech-to-text	librispeech (test-clean)	LAS (LSTM) [67, 68]	Convformer [55]	13.6% (-0.3 WER[5])
	librispeech (test-other)			25.0% (-1.3 WER)

a single model comprising several parts trained together. Zhang et al. [57] introduced an end-to-end speech recognition model based on Transformer encoders called the Transformer Transducer that outperformed previous hybrid and end-to-end approaches on the LibriSpeech benchmarks.

The Transformer's capacity comes at the cost of a quadratic computational and memory complexity with respect to the sequence length. Therefore, training large Transformers is prohibitively slow and expensive. For instance, Liu et al. [69] introduced RoBERTa, which was pre-trained on 1024 high-end V100 graphics processing units (GPUs) for approximately a day. Although numerous large pre-trained Transformers have been publicly released, fine-tuning them on the tasks of interest is still computationally expensive. Furthermore, the sequence lengths are restricted by the amount of memory available. Indeed, practitioners typically use large mini-batches with relatively short sequences because the Transformer's optimization is known to be particularly unstable with small mini-batches. Typically, a GPU with 16 GB of memory handles sequences up to 512 words. Consequently, there exists an actual need for lighter and faster Transformers as only a few large organizations can afford to train massive models. As of the writing of this paper, the largest dense Transformer is GPT-3 [50] which requires 355 years to train on a V100 GPU, costing around 4,600,000$ of cloud instances[1].

3.1.2 Lighter and Faster Transformers

Over the years, numerous approaches have been proposed to reduce the computational and memory costs of neural networks, many of which have been applied to Transformers. In this paper, such methods are referred to as *general* since they apply, and have been applied, to

[1]`https://lambdalabs.com/blog/demystifying-gpt-3`

a wide range of models. General methods are often orthogonal, and consequently, several of them may be combined to precisely fine-tune the network's capacity, computational cost, and memory usage. However, general methods may be insufficient as the model complexity typically remains unchanged. Therefore, many works introduced lower-complexity variations of the Transformer, referred to as `x`-formers. In this survey, the Transformer's alternatives are categorized depending on whether they sparsify the attention, factorize it, or modify the network's architecture. Please note that this survey aims to provide a comprehensive summary of the methods that improve the Transformer's efficiency and that fine-grained taxonomies have already been proposed by Tay et al. [70] and Lin et al. [71]. Accordingly, our taxonomy will remain purposefully coarse.

Recently, Tolstikhin et al. [72] and Liu et al. [73] amongst others argued that the powerful yet expensive self-attention mechanism is not necessary to achieve state-of-the-art results and thus challenged the preconception that the self-attention is the source of the Transformer's success. Consequently, they introduced networks without self-attention that are competitive with Transformers for image classification and language modelling at the same computational cost. Yu et al. [74] expanded on this idea with a more general and flexible architecture called MetaFormer where the mechanism to relate the tokens is not specified while the other components are kept the same as the Transformer. Despite the recent success of attention-free architectures, such networks are outside the scope of this paper as they arguably remove the Transformer's core mechanism and are discussed in the supplementary material.

The remainder of this survey is organized as follows. Section 3.2 introduces the Transformer's architecture and the origin of the quadratic complexity. Section 3.3 investigates the popular general methods that have been applied to Transformers to reduce the computations and memory footprint. Section 3.4 explores the recent lower-complexity Transformers. Section 3.5 explains the limitations of the different approaches and the current evaluation methodology, Section 3.6 provides a discussion on the broader impact of lighter and faster Transformers, and Section 3.7 points out potential future research directions. Finally, Section 3.8 concludes this survey. Practitioners and researchers can find detailed practical guidelines regarding the general and specialized methods in the supplementary material, as well as a discussion about some of the most popular attention-free alternatives.

3.2 Transformer

This section formally introduces the attention mechanism, the Transformer's architecture, and the root cause of its quadratic complexity.

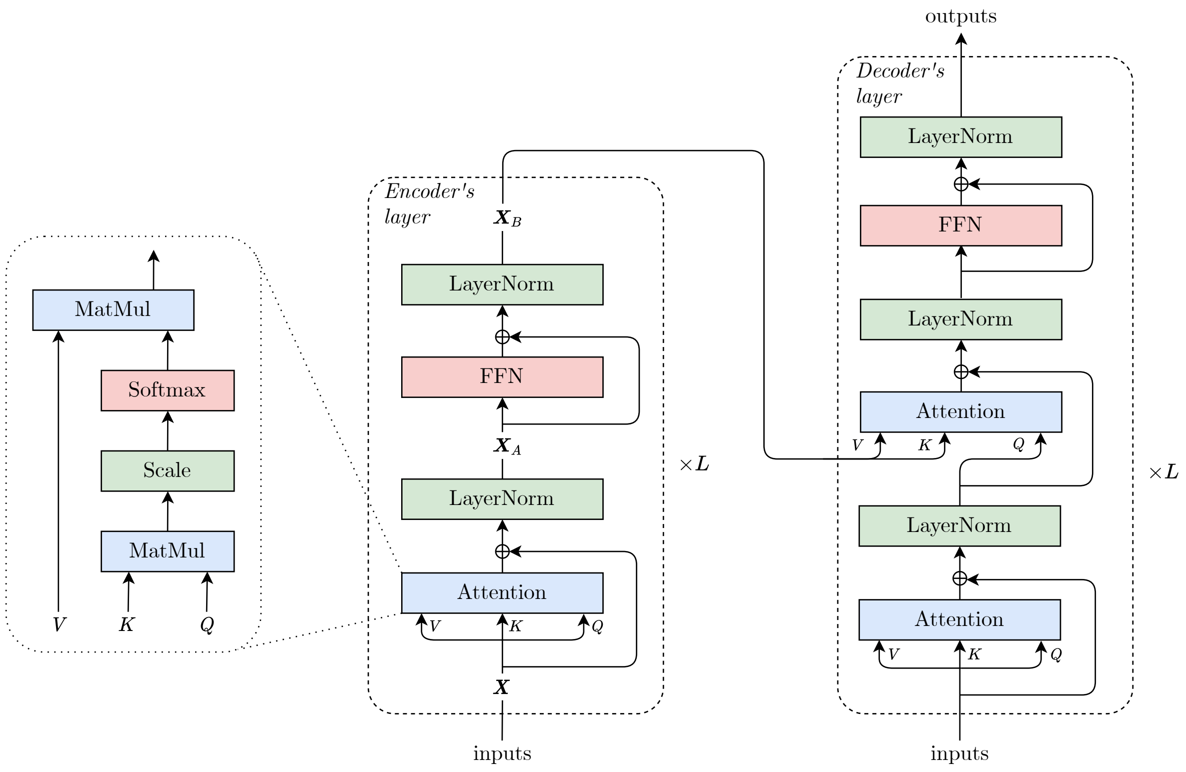


Figure 3.4 The Transformer's computational graph [43]. From left to right, the scaled dot product self-attention, the encoder, and the decoder. Note that both the encoder and decoder comprise L identical layers, of which only one is depicted.

3.2.1 Attention Mechanism

The attention mechanism relies on three matrices, namely $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{n \times d}$, commonly referred to as "queries", "keys", and "values", respectively. The attention outputs the sum of the values weighted by a compatibility or alignment score between each token, which is computed with the function $\text{Score}(\boldsymbol{Q}, \boldsymbol{K}) \in \mathbb{R}^{n \times n}$. Intuitively, if the i-th query is highly compatible with the j-th key, then the j-th value greatly contributes to the i-th attention's output. The attention mechanism may be written as:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Score}(\boldsymbol{Q}, \boldsymbol{K})\boldsymbol{V}. \tag{3.1}$$

Since the compatibility score directly controls the alignment between the tokens, many functions have been proposed. In the original paper, the Transformer relies on the *scaled dot product attention*. The *dot product* refers to the computation of the compatibility score between a single query and a single key. In practice, however, the compatibility scores are computed simultaneously for every query and key by multiplying $\boldsymbol{Q}$ with $\boldsymbol{K}^\top$. Indeed, the

(i, j) entry of the $\boldsymbol{QK}^\top$ multiplication is equal to the dot product between the i-th query and the j-th key. In order to obtain a probability distribution over the positions, referred to as attention weights, each row of $\boldsymbol{QK}^\top$ is passed through a Softmax function defined as follows:

$$\text{Softmax}(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \quad \text{for } i = 1, \dots, n. \tag{3.2}$$

where $\boldsymbol{x} \in \mathbb{R}^n$. Since the dot product grows large in magnitude for large values of d, thereby pushing the Softmax into a region of small gradients, a *scaling* factor $\sqrt{d}$ is introduced. Thus, the scaled dot product attention is given by:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Softmax}\left(\frac{\boldsymbol{QK}^\top}{\sqrt{d}}\right)\boldsymbol{V}. \tag{3.3}$$

Nonetheless, the attention presented above may not be flexible enough if the relevant information for the task is scattered across different regions of the input space. That is due in part to the Softmax being exponential, which amplifies the differences between the values. As a result, only a few attention weights are large, i.e., only a few positions are strongly attended. Vaswani et al. [43] addressed this limitation with the multi-head attention. The d-dimensional queries, keys and values matrices are first linearly projected h times with distinct, learned projections to d_k, d_k and d_v dimensions, respectively. On each projection, an independent attention instance called *head* is applied, and the output of each attention head is concatenated before being linearly projected. The Transformer's multi-head scaled dot product attention is given by:

$$\text{MultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = [\text{head}_1; ...; \text{head}_h]\boldsymbol{W}^O. \tag{3.4}$$

$$\text{head}_i = \text{Softmax}\left(\frac{\boldsymbol{QW}_i^Q(\boldsymbol{KW}_i^K)^\top}{\sqrt{d_k}}\right)\boldsymbol{VW}_i^V. \tag{3.5}$$

where $\boldsymbol{W}_i^Q \in \mathbb{R}^{d \times d_k}$, $\boldsymbol{W}_i^K \in \mathbb{R}^{d \times d_k}$, $\boldsymbol{W}_i^V \in \mathbb{R}^{d \times d_v}$ are the matrices that project the queries, keys, and values into the i-th subspace, respectively, and where $\boldsymbol{W}^O \in \mathbb{R}^{hd_v \times d}$ is the matrix that computes a linear transformation of the heads. Typically, $d_k = d/h$ where d is the input and output dimension, and h is the number of heads. For the sake of clarity, methods that modify the attention will be explained in the context of a single head (see Equation 3.3).

Thus far, the attention mechanism has been described as a general method. The Transformer relies on two specific instances of this mechanism: the intra-attention, popularly known as self-attention, and the inter-attention, sometimes referred to as cross-attention. In the case of inter-attention, the queries correspond to the decoder's hidden representations, and the keys and values are the encoder's outputs. It allows the decoder to look at the relevant parts of the input to produce the output. In the case of self-attention, the three matrices are linear projections of the layer's input, which allows the encoder and decoder to focus on the relevant part of the sequence for each position, similarly to the inter-attention depicted in Figure 3.3.

3.2.2 Encoder

The Transformer's encoder is a function defined as the composition of L identical layers or blocks, each composed of two sub-layers. The first sub-layer is the aforementioned self-attention mechanism. The second sub-layer is a simple fully connected feed-forward network applied position-wise, that is, independently and identically to every position. The feed-forward network increases the encoder's expressiveness and transforms the self-attention's output for the next layer.

Inspired by ResNet [75], a skip connection, shortcut connection, or residual connection is applied around each sub-layer to create a direct path for the gradient to flow throughout the network. Notably, residual connections make the training of very deep neural networks more stable. Additionally, both sub-layers' outputs are normalized after the residual connection with the layer normalization technique, referred to as LayerNorm [76]. Normalization is a widely adopted technique in deep learning that enables faster and more stable training. Although the rationale behind the normalization's empirical success is not yet fully understood [77], it has been conjectured that this results from a smoother optimization landscape, and to a lesser extent, from a reduction in internal covariance shift [78]. Figure 3.4 depicts the computational graph of an encoder's layer.

In natural language processing, the input sequence $\boldsymbol{X}$ would typically represent a sentence or a paragraph, and the token $\boldsymbol{x}^{(i)}$ would be its i-th word or subword embedding. Each encoder's layer is given by:

$$\boldsymbol{X}_A = \text{LayerNorm}(\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) + \boldsymbol{X}) \tag{3.6}$$

$$\boldsymbol{X}_B = \text{LayerNorm}(\text{FFN}(\boldsymbol{X}_A) + \boldsymbol{X}_A) \tag{3.7}$$

where $\boldsymbol{X}$ and $\boldsymbol{X}_B$ are the layer's input and output, respectively, and $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ are linear projections of $\boldsymbol{X}$.

The feed-forward network is given by:

$$\text{FFN}(\boldsymbol{x}) = \max(0, \boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2 \tag{3.8}$$

where $\boldsymbol{W}_1 \in \mathbb{R}^{d \times d_f}$ and $\boldsymbol{W}_2 \in \mathbb{R}^{d_f \times d}$, and where d_f is the dimension of the hidden layer. Note that the feed-forward network is defined for a row vector since it is applied position-wise, that is, it is independently and identically applied to every position or row.

Finally, the position-wise layer normalization is given by:

$$\text{LayerNorm}(\boldsymbol{x}) = \boldsymbol{g} \odot \frac{\boldsymbol{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \boldsymbol{b} \tag{3.9}$$

where $\odot$ denotes the element-wise (Hadamard) product, where the average μ and the standard deviation σ are computed from all of the summed inputs, where the gain $\boldsymbol{g}$ and the bias $\boldsymbol{b}$ are learned parameters of dimension d, and where ϵ is a small constant used in practice for numerical stability.

3.2.3 Decoder

The decoder is also composed of L identical layers. Although it is common for the decoder to have the same number of layers as the encoder, one may adjust their depth independently. Each decoder's layer comprises three sub-layers. The first sub-layer is the self-attention mechanism, as in the encoder, except that future positions are masked. Indeed, while the encoder is allowed to look at future positions since the input sequence is entirely available, the decoder is autoregressive and thus cannot look at future positions since they have not yet been predicted. Therefore, the i-th position may only attend to positions less than i. The second sub-layer is the inter-attention mechanism, which helps the decoder focus on the relevant parts of the input. Finally, the third sub-layer is a simple feed-forward network. As for the encoder, a residual connection and a layer normalization are applied to each sub-layer.

Note that the decoder may be safely omitted when the task does not require the sequence-to-sequence framework, such as sentiment analysis, which predicts whether a sentence is positive. One of the most popular encoder-only Transformers is the Bidirectional Encoder Representations from Transformers (BERT) [51], a state-of-the-art language model that learns contextualized embeddings. Nonetheless, autoregressive tasks such as machine translation still require the sequence-to-sequence framework.

3.2.4 Complexity

Intuitively, the quadratic complexity emerges from the computation of the compatibility score between every pair of positions. More precisely, the $\boldsymbol{QK}^\top$ multiplication requires n^2 computations and memory. Such attention is said to be full since any output position is able to attend to any input position. The attention pattern is visualized by means of a connectivity matrix, which indicates the input positions that each output position is able to attend (see Figure 3.5).

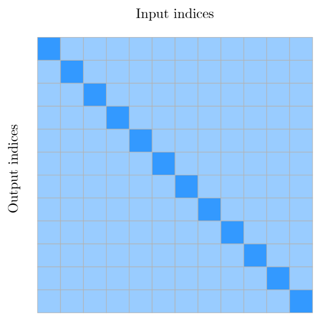


Figure 3.5 The connectivity matrix of the full attention. The i-th output position attends to the j-th input position if, and only if, the cell (i, j) is coloured. The diagonal is highlighted to ease the reading.

What justifies such efforts from the community to improve the Transformer's efficiency? In our opinion, there are three primary motivations: affordability, scalability, and ecology.

The foremost reason is affordability. The Transformer has largely surpassed convolutional and recurrent neural networks and achieved new state-of-the-art results across many tasks. However, those networks have a linear complexity with respect to the sequence length [43], making them affordable to most researchers and practitioners. As explained by Strubell et al. [79], this creates three major issues: (1) it stifles creativity as researchers and practitioners that do not have access to considerable resources are not able to experiment with Transformers, (2) it reinforces the "rich get richer" cycle where successful labs and companies receive more funding due to their existing accomplishments with Transformers, and (3) it forces smaller labs and companies to rely on private cloud services that end up more expensive.

The second reason is scalability. The quadratic complexity prevents researchers and practitioners, even those with access to considerable resources, from applying Transformers on long sequences such as entire chapters or books, high-resolution images or videos, and DNA.

The third reason is ecology. It is now more apparent than ever that we must cut carbon dioxide (CO2) emissions in half over the next decade to limit global warming. The large-scale infrastructures used by the deep learning community consume a considerable amount of electricity, which is mainly produced by non-renewable sources such as coal or gas [80].

Thereby, the following sections investigate popular and novel methods to make Transformers faster and lighter.

3.3 General Approaches

Computational resources have always been a limiting factor for deep learning models [81]. Therefore, numerous approaches have been proposed throughout the years to design faster and lighter models. This section introduces the most popular techniques that apply to virtually all neural networks.

Gradient Checkpointing [82]: Intermediate results computed during the forward pass, also referred to as activations, are required to compute the gradients during the backward pass; therefore, they are stored in memory. Activations typically account for most of the memory during training: given an l-layer network, the number of intermediate results is proportional to the number of layers ($\mathcal{O}(l)$). With gradient checkpointing, also known as rematerialization, activations are stored only for a subset of the layers. However, they must be recomputed during the backward pass, trading memory for computations. In the extreme case where no activations are stored, the memory usage becomes constant ($\mathcal{O}(1)$) at the cost of a quadratic number of computations with respect to the number of layers ($\mathcal{O}(l^2)$). Chen et al. [82] designed a scheme to select the preserved values that reduces the memory requirement from $\mathcal{O}(l)$ to $\mathcal{O}(\sqrt{l})$ at the cost of a single additional forward pass per mini-batch. OpenAI implementation of gradient checkpointing [83] obtains an impressive 10$\times$ reduction in memory at the cost of a 20% increase in computation time.

Reversible Layers [84, 80, 85, 86]: As explained above, the back-propagation requires the activations of all intermediate layers, which are either stored in memory during the forward pass or recomputed during the backward pass. As a solution to the latter case, reversible layers allow their activation to be reconstructed exactly from the next layer; therefore, activations must only be stored for one layer and their memory cost becomes independent of the network's depth. More formally, each reversible layer takes as input (x_1, x_2) and outputs (y_1, y_2) such that $y_1 = x_1 + f(x_2)$ and $y_2 = x_2 + g(y_1)$. Each layer's activations are easily reconstructed as $x_2 = y_2 - g(y_1)$ and $x_1 = y_1 - f(x_2)$.

Kitaev et al. [87] used reversible layers in their Transformer, called the Reformer, by combining the attention and feed-forward sub-layers inside a reversible layer. Specifically, $f(.)$ and $g(.)$ were the Attention(.) and FFN(.) functions, respectively. The authors observed that reversible layers reduced the memory usage of a 3-layer Transformer without degrading its performance. Nonetheless, reversible layers add numerical errors that accumulate over multiple layers and may degrade the model performance. Therefore, they are not suited for very deep networks.

Gradient checkpointing and reversible layers are very much alike in that they trade computations for memory by recomputing activations during backpropagation. This trade-off is sometimes necessary: although computation bottlenecks entail longer running times, memory bottlenecks are critical as they prevent using the model altogether.

Parameter Sharing: A simple approach to reduce the number of trainable parameters is to impose sets of parameters to be equal in different parts of the network. In other words, the same parameters are used for multiple operations but need to be stored only once in memory. Such a technique is often referred to as parameter sharing, weight tying, or weight replication. As explained in Section 3.1 and illustrated in Figure 3.1, recurrent neural networks are built around this idea of parameter sharing to process variable-length sequences. Parameter sharing has also been applied to Transformers. For instance, the Linformer [88] shares projection matrices across heads and layers, and the Reformer [87] shares its queries and keys parameters, that is, $\boldsymbol{W}^Q = \boldsymbol{W}^K$. Both authors investigated the impact of parameter sharing and concluded that it did not degrade their respective models' performance on their tasks. Lan et al. [89] shared all parameters between layers, which drastically reduced the number of parameters but also decreased the performance by up to 2.5% on average. They observed that sharing only the attention parameters resulted in a slight drop in performance of 0.7% on average. The decrease in performance is to be expected since parameter sharing reduces the number of free parameters, hence the model's capacity.

Pruning [81]: Smaller neural networks are not only faster and lighter, but they are also more likely to generalize better than larger models because they presumably extract underlying explanatory factors without redundancy. To reduce the model size, weights with a small saliency, that is, whose deletion have a small effect on the loss, may be removed from large models after training. Methods that consider individual weights are said to be *unstructured*, and methods that consider pieces of the network structure such as attention heads or layers are said to be *structured*. Many structured and unstructured pruning schemes have been proposed, several of which have been applied to Transformers. For instance, Sajjad et al. [90] reduced the size of BERT by 40% by dropping complete layers while retaining between 97 and 98% of its original performance, and Michel et al. [91] pruned away between 20% and 40% of BERT attention heads without any significant loss in performance. Recently, the lottery ticket hypothesis has brought a new justification to pruning neural networks. As introduced by Frankle and Carbin [92], the hypothesis states that a "*randomly-initialized, dense neural network contains a subnetwork that is initialized such that – when trained in isolation – it can match the test accuracy of the original network after training for at most the same number of iterations.*". Prasanna et al. [93] successfully verified this hypothesis on BERT, even noticing that BERT worst subnetworks remain highly trainable. Nonetheless, pruning has two limitations: a large model must be trained, and unstructured pruning schemes produce sparse models unoptimized for modern GPUs and tensor processing units (TPUs).

Knowledge Distillation [94, 95]: The knowledge of a large model or an ensemble of models (teacher) is transferred to a single smaller model (student) by training the student to reproduce the teacher's outputs or its internal behaviour. The cumbersome teacher is then discarded, and the student is used at inference time. Given a parameter budget, networks trained with knowledge distillation usually outperform models directly trained on the task. Sanh et al. [96], Tsai et al. [97], and Jiao et al. [98] applied different knowledge distillation schemes on the original BERT [51] to obtain lighter and faster models called DistilBERT, MiniBERT, and TinyBERT, respectively. Table 3.2 reports their compression, speed-up, and performance. Although knowledge distillation achieves impressive compression ratios and performance trade-offs, a large teacher model still needs to be trained, and the student may perform significantly worse than the teacher. For instance, BERT$_{\text{BASE}}$ achieves an accuracy of 52.8% on the CoLA task [99], while DistilBERT and TinyBERT only achieve 32.8% and 44.1%, respectively, according to Jiao et al. [98].

Table 3.2 Multiple knowledge distillations of BERT$_{\text{BASE}}$. Speed-ups are evaluated on GPUs.

Model	Compression	Speed-up	Mean Relative Performance
BERT$_{\text{BASE}}$ [51]	1.0×	1.0×	100%
DistilBERT [96]	1.7×	1.6×	97%
MiniBERT [97]	6.0×	2.6 – 4.3×	97 – 99%
TinyBERT [98]	7.5×	9.4×	97%

Mixed-Precision [100]: Modern GPUs and TPUs perform at least twice as many half-precision (16 bits) float operations as single-precision (32 bits) ones. A popular approach to accelerate training and reduce memory consumption is storing and computing the weights, activations, and gradients in half-precision. A master copy of the weights is stored in single-precision for numerical stability and minimal performance loss. Thanks to NVIDIA's Automatic Mixed-Precision included in some of the most popular deep learning libraries, namely TensorFlow, PyTorch, and MXNet, using mixed precision can be as simple as adding one line of code. Consequently, we highly recommend mixed-precision. Jacob et al. [101] improved over this approach by quantizing both weights and activations as 8-bit integers and biases as 32-bit integers, effectively allowing inference to be performed using integer-only arithmetic. Given a parameter matrix $\boldsymbol{W}$, N-bit quantization rounds each parameter to one of 2^N codewords corresponding to bins evenly spaced by a scale factor s and shifted by a bias z computed as follows:

$$s = \frac{\max \boldsymbol{W} - \min \boldsymbol{W}}{2^N - 1} \quad \text{and} \quad z = \text{round}\left(\frac{\min \boldsymbol{W}}{s}\right) \tag{3.10}$$

Each parameter $W_{i,j}$ is quantized to its nearest codeword, and dequantized as:

$$\hat{W}_{i,j} = \left(\text{round}\left(\frac{W_{i,j}}{s} + z\right) - z\right) \times s \tag{3.11}$$

In order to mitigate the performance loss associated with the low-precision approximation, Quantization Aware Training (QAT) [101] quantizes the parameters during training. Since quantization is not differentiable, gradients are approximated with a straight-through approximator [102]. Notably, Zafrir et al. [103] quantized all matrix product operations in BERT fully connected and embedding layers during training, reducing the memory footprint by 4× while retaining 99% of the original accuracy on the GLUE [66] and SQuAD [104] tasks. Stock et al. [105] achieved an even higher compression ratio with iterative product quantization (iPQ), which replaces vectors of weights by their assigned centroid, and quantization of those centroids. The authors reduced the size of a 16-layer Transformer by 25×, making the model only 14 MB, while retaining 87% of the original performance on the Wikitext-103 [106] benchmark.

While pruning and knowledge distillation achieve faster and lighter models by reducing the number of parameters, mixed-precision and quantization instead reduce the number of bits per parameter.

Micro-Batching [107]: Increasing model capacity and data throughput are efficient strategies for improving performances in deep learning. However, increasing data throughput requires transferring large mini-batches to the accelerators' memory[6], which is also used to store the model. One way to partially avoid the trade-off between mini-batch size and model size is to use model parallelism. GPipe [107] is a model parallelism library that enables users to distribute a model by grouping layers into cells assigned to accelerators. To avoid the communication bottleneck between accelerators due to the forward and backward operations, the authors proposed a novel batch-splitting algorithm that further splits the mini-batch into micro-batches. As soon as the first accelerator finishes the forward operation of the layers assigned to it for a micro-batch, it sends the result over the communication link and starts processing the next micro-batch. After finishing the last micro-batch's forward operation, the accelerators wait for the first micro-batch's backwards operation results. This waiting time can be used to recompute the forward operation and further reduce memory usage, known as rematerialization. Finally, once the backward operation is completed on the last micro-batch, the algorithm sums all micro-batch's gradients to obtain the mini-batch's gradient (see Figure 3.6). However, the result is not exact with layers that compute statistics

[6]An accelerator denotes any device that accelerates computation, such as a graphics or tensor processing unit.

across all mini-batch examples, such as a batch normalization layer [108]. Finally, GPipe is compatible with data parallelism, where multiple mini-batches are processed in parallel.

Huang et al. [107] empirically demonstrated that GPipe allows the maximum Transformer size to scale linearly with the number of accelerators. For instance, a TPU v3 with 16Gb of memory can only fit a 3-layer Transformer. With GPipe, the same TPU is able to fit 13 layers, while 128 TPUs are able to fit 1663 layers, which is 127.9× more. Additionally, the authors distributed a 48-layer Transformer across 8 TPUs and reported that the training throughput was 4.8 times higher with 32 micro-batches than with a single one.

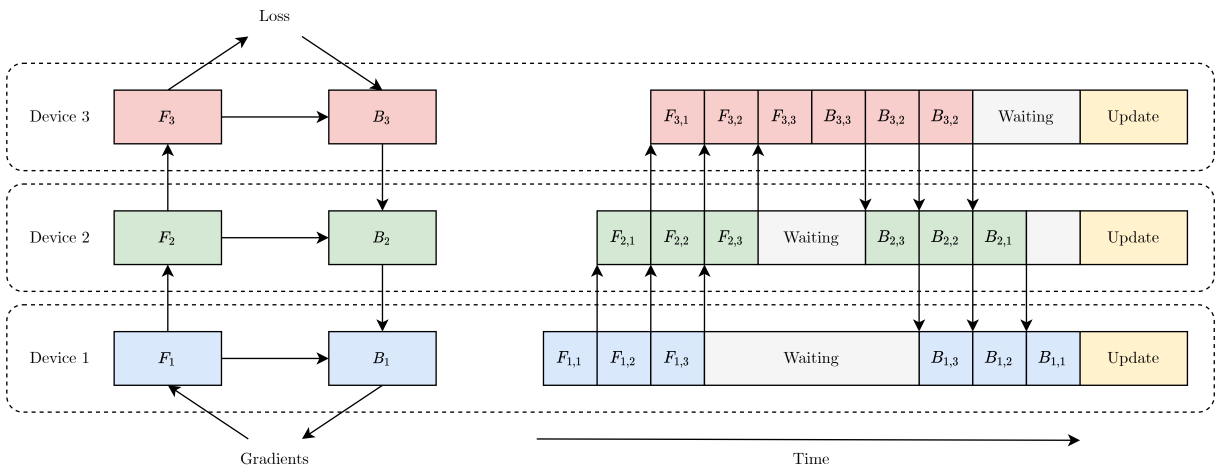


Figure 3.6 Micro-Batching applied to a model distributed across three devices [107]. F_i and B_i denotes the sequential forward and backward operations, respectively, performed by the i-th device. Computation on a device may start as soon as the previous device in the computational graph has processed the first micro-batch. Therefore, micro-batching reduces the waiting time of each device at the cost of inter-device communications. Note that the model update is done synchronously at the end.

Mixture of Experts [109]: The core idea is to train multiple networks called experts, each of which specializes only in a subset of the data, and a manager or router, which forwards the input to the corresponding experts. A single network is used in practice, whose layers are composed of multiple subsets of parameters (experts), effectively resulting in a sparsely activated model as illustrated in Figure 3.7. Increasing the number of experts keeps the computational cost constant since the model always selects the same number of experts for each input regardless of the number of experts. Therefore, the mixture of experts (MoE) approach allows for massive models and is particularly efficient for distributed systems in which experts are spread across devices. In that case, the number of experts, and therefore parameters, scales with the number of devices available. Despite these advantages, the mixture of experts has not yet been widely adopted as the method is complex to deploy in practice. It imposes a communication cost between the devices, a computation cost to select the experts for each input position, and makes training unstable. Recently, Fedus et al. [110] introduced the Switch Transformer based on a carefully crafted mixture of experts. Notably, given a fixed amount of computation per input position, the Switch Transformer reached the same quality threshold as a vanilla Transformer five times faster (wall-clock time) on average. Additionally, when trained further, the Switch Transformer outperformed the vanilla baseline. However, this approach assumes that multiple regimes with distinct input to output relations produce the data.

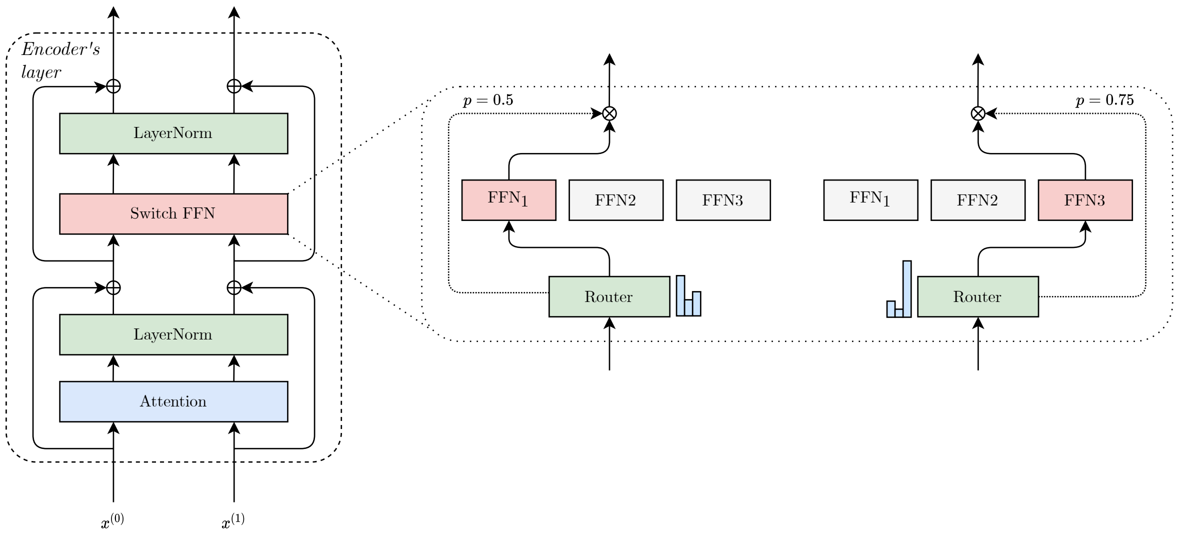


Figure 3.7 The computational graph of a single layer of the Switch Transformer's encoder [110]. The Transformer's feed-forward network (FFN) has been replaced by a Switch FFN which independently routes each position to an expert. The expert's output is multiplied by the gate value. Note that the computational cost is independent of the number of experts since a single expert is active for each position.

Difficult tasks often require large models to achieve the desired performance. However, such models require powerful and expensive accelerators. Both micro-batching and the mixture of experts offer an alternative to train such models on many relatively weak and inexpensive GPUs at the cost of complex implementation.

Sample-Efficient Objective [111]: Large neural networks, especially Transformers, benefit from being pre-trained with an unsupervised objective before being fine-tuned on the task of interest, also called the downstream task. The core idea is to leverage large unlabelled datasets that are easy to automatically collect in order to learn the data underlying explanatory factors and ultimately improve the model performance. Concretely, pre-training initializes the network's weights in a "good" region of space. As pre-training of large models is often more compute-intensive than fine-tuning, researchers regularly share pre-trained models to facilitate their adoption. Most notably, Hugging Face [112] is an open-source library that contains an extensive collection of pre-trained Transformers under a unified API. Nonetheless, researchers must sometimes pre-train models themselves due to the peculiar nature of the data or the problem at hand. In that case, a sample-efficient objective will reduce the computation required.

Recently, Devlin et al. [51] popularized the Cloze procedure [113] for pre-training under the name of masked language model (MLM), which independently estimates the probability of masked words given the rest of the sequence. Practically, 15% of the words are randomly selected, of which 80% are masked, 10% are replaced by a random word, and 10% are left unchanged. This task is analogous to the reconstruction of corrupted input. Figure 3.8 illustrates the masked language model objective.

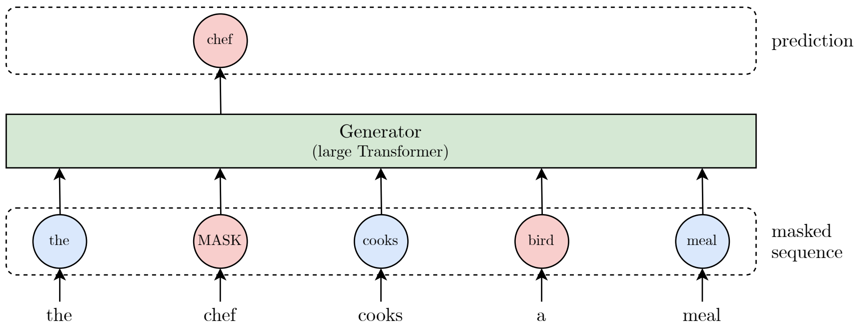


Figure 3.8 The masked language model objective [51]. The masked words are depicted in red. The model makes a prediction only for the masked words; thus, MLM is computationally inefficient.

Clark et al. [111] introduced the replaced token detection objective to speed up pre-training; a small network (generator) first generates a plausible alternative for each masked word, then the large model (discriminator) predicts whether each word has been replaced (see Figure 3.9). While the masked language model makes a prediction only for the masked works, the replaced token detection makes a prediction for every word. Therefore, the latter is more computationally efficient than the former; in other words, less pre-training computations are required to achieve the same performance on downstream tasks. Additionally, the authors reported that the representations learned with their objective outperformed those learned with MLM given the same model size, data, and computation. Most notably, they were able to outperform GPT on the GLUE benchmark with 30× fewer computations.

Parameter Initialization Strategies: Optimizing deep networks is challenging in part

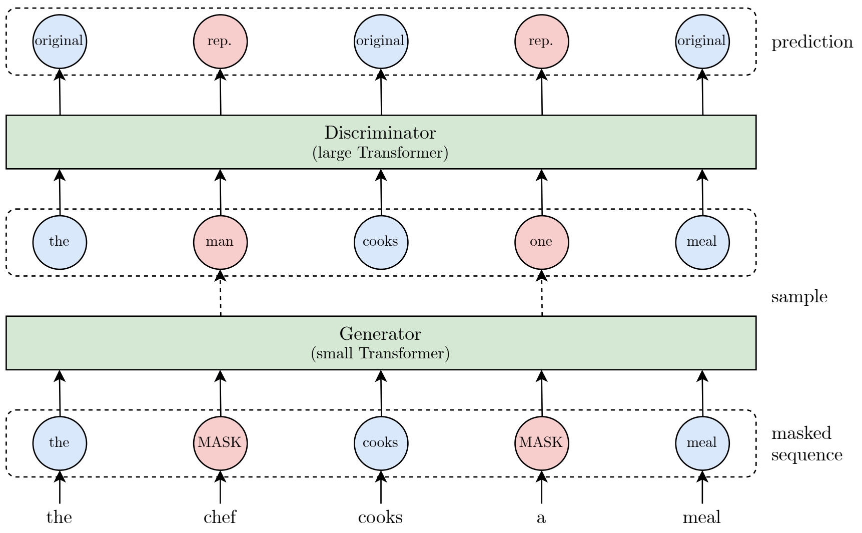


Figure 3.9 The replaced token detection objective [111]. A plausible alternative of each masked word is sampled from a small generator network. Then a discriminator predicts whether each word has been replaced.

because of the considerable influence of the initial point on the iterative process. Notably, the initial point determines whether the algorithms converge at all and, if it does converge, the speed at which it converges as well as the quality of the solution [28]. Transformers are notoriously difficult to train, typically requiring carefully tuned optimizers with adaptive learning rates, learning rate schedulers, and large batches. Even then, convergence is not guaranteed. Consequently, Liu et al. [114] and Huang et al. [115] concurrently proposed initialization schemes for the Transformer that promise a smoother and faster optimization as well as better generalization performances.

Liu et al. [114] identified an amplification effect that significantly influences training: each layer heavily depends on its residual branch[7], making the optimization unstable as it amplifies small parameter perturbations. Ultimately, the amplification effect may produce a notable change in the Transformer's output. Nonetheless, the authors observed that heavy dependencies on the residual branches are necessary to unlock the Transformer's potential and achieve better results. In order to mitigate the amplification effect, Liu et al. [114] introduced the Adaptive Model Initialization strategy, or Admin, that controls the dependency on the residual connections in the early stage of training with a new parameter $\boldsymbol{\omega}$. Formally,

[7]For a residual block $f(x) + x$, the residual branch refers to $f(x)$ and the skip connection, shortcut connection, or residual connection refers to x.

the i-th sub-layer output is given by

$$\boldsymbol{X}_i = \text{LayerNorm}(f_i(\boldsymbol{X}_{i-1}) + \boldsymbol{X}_{i-1} \odot \boldsymbol{\omega}_i), \tag{3.12}$$

where $f_i(\boldsymbol{X})$, $\boldsymbol{X}_{i-1}$, and $\boldsymbol{X}_i$, denote the function, input, and output of the i-th sub-layer, respectively. Although this is equivalent to rescaling some model parameters, the authors observed that rescaling leads to unstable training in half-precision.

The proposed initialization strategy requires three steps. First, the model parameters are initialized with a standard method such as the Xavier initialization [116] and the Admin parameter $\boldsymbol{\omega}$ with ones. Then, one or a small number of mini-batches are forward propagated without updating the parameters and record the output variance of each residual branch $\text{Var}[f_i(\boldsymbol{X}_{i-1})]$. Finally, the Admin parameter is initialized as $\boldsymbol{\omega}_i = \sqrt{\sum_{j<i} \text{Var}[f_j(\boldsymbol{X}_{j-1})]}$. Once the model has been trained, $\boldsymbol{\omega}$ may be discarded.

The amplification effect is, however, not the only mechanism that makes Transformers notoriously difficult to train. Huang et al. [115] addressed two other issues: (i) Transformers are typically trained with optimizers that rely on adaptive learning rates as conventional SGD fails to train them effectively. However, adaptive learning rates have a problematically large variance in the early stages of optimization, resulting in convergence issues [117]; and (ii) the magnitude of the error signal propagated through LayerNorm is inversely proportional to the magnitude of the input [118]. Specifically, the norm of the layer normalization gradient is proportional to:

$$\left\| \frac{\partial \text{LayerNorm}(\boldsymbol{x})}{\partial \boldsymbol{x}} \right\| = \mathcal{O}\left(\frac{\sqrt{d}}{\|\boldsymbol{x}\|} \right) \tag{3.13}$$

Consequently, if the input norm $\|\boldsymbol{x}\|$ is larger than $\sqrt{d}$, backpropagating through layer normalization reduces the gradient magnitude for layers closer to the input. As a solution to both problems, Huang et al. [115] proposed an initialization strategy called T-Fixup that restricts the magnitude of the updates in the early stages of training, thus mitigating the vanishing gradient issue while eliminating the need for layer normalization and warmup.

While Liu et al. [114] and Huang et al. [115] claim faster convergence, they omitted to report the improvement.

Architecture Search: One of the most challenging goals in deep learning is to automatically design networks. Indeed, the problem of finding architectures that achieve the best performance with the fewest operations and lowest memory footprint in a discrete search space is an NP-hard combinatorial optimization problem. Over the years, multiple approaches to Neural Architecture Search (NAS) have been proposed, including reinforcement learning [119], evolutionary algorithms [120], and bilevel optimization [121]. Notably, Zoph et al. [122] demonstrated that NAS is able to surpass human-designed architectures on ImageNet by 1.2% top-1 accuracy while using 28% fewer computations. Nonetheless, neural architecture search methods are computationally expensive as they usually require training each candidate model from scratch. As a solution, Pham et al. [123] proposed Efficient NAS (ENAS), which constrains all candidates to be subgraphs of a single computational graph, that is, to share parameters. Therefore, the ENAS's controller decides which operations are activated and relies on the models' ability to adapt, similarly to dropout [124]. Efficient NAS reduces the search computational budget by 1,000× over the original NAS [119]. Alternatively, Liu et al. [121] proposed the Differentiable Architecture Search (DARTS), which casts the NAS problem as a differentiable bilevel optimization problem. The first level consists of a continuous relaxation of the discrete search space using a Softmax function over a list of candidate operations, and the second level involves the model's weights. However, the bilevel formulation requires training the weights to convergence to evaluate the architecture gradient. To avoid this substantial cost, the authors made the approximation of taking a single gradient step of the weights for one gradient step of the architecture parameters. The authors obtained comparable performances to non-differentiable NAS methods on ImageNet in the mobile setting using only 4 GPU-days, compared to 3,150 for evolutionary algorithms [120] and 2,000 for NAS [122]. Differentiable Architecture Search obtained comparable results to ENAS with a similar computational budget. We refer the reader to Elsken et al. [125] survey for further detail on architecture search methods.

Nevertheless, neural architecture search methods are challenging to apply on Transformers due to the memory requirements and training time. Therefore, recent works introduced methods better suited for the Transformer. So et al. [126] modified the tournament selection evolutionary architecture search [120] with Progressive Dynamic Hurdles (PDH), which dynamically allocates resources to more promising architectures according to their performances. With PDH, the authors optimized transformer architectures directly on the WMT'14 En-De task [127] which requires 10 hours of computation on a Google TPU v2 for the base Transformer model. Training directly on this dataset is essential since the authors did not find a smaller surrogate dataset that transfers well, such as CIFAR-10 for ImageNet. The Evolved Transformer matched the vanilla Transformer's performance with only 78% of its parameters. Recently, Tsai et al. [128] profiled the Transformer's components on a TPU v2 and observed that some mechanisms substantially impact inference time: attention queries, keys, and values dimensions, width and depth of feed-forward layers, number of attention heads, and layer normalization mean computation. By decomposing these components into building blocks and using binary variables, the authors perform a one-shot search for both the architecture and the parameters with a single loss. They optimized this loss with gradient descent on a continuous relaxation of the binary variables and used policy gradient algorithm. Tsai et al. [128] were able to make miniBERT 1.7× faster with a performance drop smaller than 0.3%. Compared to the original BERT, this is 33 to 36× faster.

Neural architecture search is a promising tool to design lighter and faster Transformers automatically. Nonetheless, NAS imposes a high computational and memory cost, which may be avoided by carefully engineering the architecture instead. For instance, the Lite Transformer [129] leverages the Long-Short Range Attention (LSRA), where a convolutional layer is applied in parallel to the self-attention in order to learn the local dependencies separately. The carefully handcrafted Lite Transformer outperforms the Evolved Transformer [126] for the mobile NLP setting while requiring about 14,000× less GPU time.

Conditional Computing [130]: Although large models are necessary for hard examples, smaller models are likely to perform as well, if not better, on simpler ones. For instance, many words such as "car" are easy to translate, while a few such as "can" require careful consideration of the context[8]. As of this survey's writing, most architectures apply a fixed number of operations to all examples regardless of their difficulty. A more efficient approach would be to reduce the amount of computation for simple examples. As a solution, Bengio [130] introduced conditional computing, which dynamically adapts the model's computational graph as a function of the input.

One way to implement conditional computing is with a mixture of experts, as introduced previously. In that case, only a subset of the parameters is used for a given input, making the computational graph sparse and the computation time almost constant with respect to the model size. Another approach consists of keeping the number of parameters constant and letting the model adjust its computation time separately for each input (according to the input's value). This approach is called Adaptive Computation Time (ACT) [131] and uses a recurrent mechanism to transform the representations until a halting probability exceeds a given threshold. The model learns to control this probability to minimize both the prediction error and the number of iterations, called the *ponder cost*, which prevents the model from using an infinite amount of computation before making a prediction. One shortcoming of the Adaptive Computation Time is its sensitivity to the ponder cost, which controls the trade-off between speed and accuracy.

Dehghani et al. [132] applied ACT to a Transformer with a recurrent mechanism for the architecture's depth. To implement this mechanism, the authors defined encoder and decoder

[8]Depending on the context, the word "can" has various meanings, including "be able to", "may", "jail", and "metal container". See `https://www.wordreference.com/definition/can`.

blocks similar to the original Transformer, except that each block is recurrent, sending its output back as its input until the ponder cost becomes too high. Note that a fixed number of recurrent steps is equivalent to a Transformer with tied parameters across all layers. With this new architecture called Universal Transformer, the authors claimed that it is computationally universal (Turing-complete) given enough memory. This property may help Transformers generalize to sequences longer than the ones seen during training. The authors obtained state-of-the-art results on algorithmic and language understanding tasks. ACT and the Universal Transformer apply the same layers iteratively, which may not be sufficiently flexible. Elbayad et al. [133] addressed this limitation with the Depth-Adaptive Transformer (DAT), which applies different layers at every depth. The DAT matches the performance of a well-tuned Transformer baseline while reducing the computation by up to 76%. However, the authors did not provide a comparison between the Universal Transformer and DAT.

In the same way that complex examples may require more computations, some may require access to a longer context. As a solution, Sukhbaatar et al. [134] dynamically adjusted the attention span, that is, the context length, by learning to mask the compatibility scores depending on the input. Their approach achieved state-of-the-art on `text8` and `enwik8` [135] while requiring significantly fewer computations. Alternatively, Li et al. [136] introduced the Decoder-end Adaptive Computation Steps (DACS), which monotonically computes halting probabilities along with the encoder states and stops the decoder computations in order to produce an output when the accumulation of probabilities exceeds a given threshold. In other words, each decoder step only looks at the necessary information as measured by the halting probabilities instead of looking at the entire input sequence.

3.4 Specialized Approaches

Since the Transformer's quadratic complexity comes from the attention mechanism, most specialized methods rely on a fast and light approximation of the original full attention. As will be explained in greater detail in the rest of this section, the attention weight matrix is dominated by a few large values and is approximately low-rank. These observations justify two distinct lines of work: sparse attention and factorized attention. Alternatively, the complexity may be reduced without altering the original attention mechanism and thus the Transformer's capacity by directly modifying the network's architecture. Let us first investigate the approaches that rely on sparse attention.

Note that some approaches only consider autoregressive tasks, such as the left-to-right language model, and in that case, the connectivity matrix is lower triangular as it is not permitted to attend to future positions. Whenever possible, such works have been extended to the more general case where attending to future positions is allowed in order to ease the comparison between the different approaches.

3.4.1 Sparse Attention

Due to the exponential nature of the Softmax, only a few positions are strongly attended to. Consequently, a conceptually simple way of reducing the Transformer's complexity is to make the matrix $\boldsymbol{QK}^\top$ sparse[9], in other words, to only allow each position to attend to a subset of the positions. Let us investigate sparse patterns that are (i) fixed and random, (ii) learned and adaptive, and (iii) identified with clustering and locality sensitive hashing.

Fixed and Random Sparse Patterns [137, 138, 139, 140, 141, 142, 58]: One of the first models to consider fixed sparse patterns is the Star-Transformer introduced by Guo et al. [137], which reduced the complexity from quadratic to linear by only allowing attention between adjacent positions. In order to preserve the Transformer's ability to model long-term dependency, the authors relied on a single global token. Global tokens, also known as shared relay nodes, can attend to every position, and every position can attend to global tokens. Let us assume that the global token is located at position 0. The i-th output position is allowed to attend to every input position if $i = 0$, otherwise, it is allowed to attend to the j-th input positions for $j = 0$ and if $i - 1 \leq j \leq i + 1$. Figure 3.10 illustrates the Star-Transformer attention pattern.

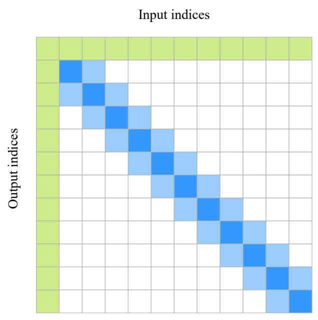


Figure 3.10 The connectivity matrices of the Star-Transformer [137].

Concurrently, Child et al. [138] introduced the Sparse Transformer which reduced the complexity to $\mathcal{O}(n\sqrt{n})$ with two different sparse attention patterns: strided and fixed. Strided attention allows the i-th output position to attend to the j-th input position if one of the two following conditions is satisfied: $(i + s) > j > (i - s)$ or $(i - j) \bmod s = 0$, where the stride

[9]Since the matrix $\boldsymbol{QK}^\top$ is passed through a Softmax function, the masked values are set to minus infinity, effectively setting their contribution to $e^{-\infty} = 0$.

s is chosen to be close to $\sqrt{n}$. Similarly, fixed attention allows i to attend to j if one of the two following conditions is satisfied: $\text{floor}(j/s) = \text{floor}(i/s)$ or $(j \bmod s) \geq (s - c)$, where c is an hyperparameter. Figure 3.11 illustrates the strided and fixed attention patterns.

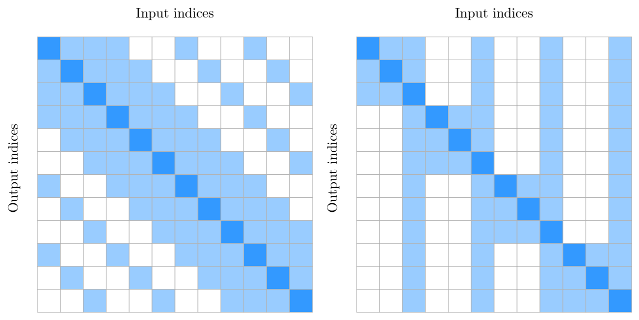


Figure 3.11 The connectivity matrices of the Sparse Transformer Child et al. [138]. (Left) Strided attention with a stride of 3. (Right) Fixed attention with a stride of 3 and $c = 1$.

Alternatively, Wang et al. [139] introduced the Cascade Transformer, which relies on sliding window attention whose size grows exponentially with the number of layers. More specifically, the number of cascade connections at the layer l is equal to $2.b.m^l - 1$, where b is the base window size and m is the cardinal number; therefore reducing the complexity to $\mathcal{O}(n.b.m^l)$. Cascade attention is well suited for shallow networks, but its complexity tends to that of the full attention in deep networks as depicted by the connectivity matrices in Figure 3.12.

Li et al. [140] introduced the LogSparse-Transformer for forecasting fine-grained time series with strong long-term dependencies. The LogSparse-Transformer relies on the eponym attention that allows the i-th output to attend to the j-th inputs for $j \in \{-2^{\lfloor \log_2 i \rfloor}, i - 2^{\lfloor \log_2 i \rfloor - 1}, \ldots, i - 2^1, i - 2^0, i, i + 2^0, i + 2^1, \ldots, i + 2^{\lfloor \log_2 (n-i) \rfloor - 1}, i + 2^{\lfloor \log_2 (n-i) \rfloor}\}$ where $\lfloor . \rfloor$ denotes the floor operation and N denotes the sequence length. Figure 3.13 illustrates the connectivity matrix of the LogSparse attention. Since only $O(\log n)$ positions are attended to by each of the n positions, the complexity of the LogSparse attention is $O(n \log n)$. Additionally, the authors proposed two alternatives: (1) to allow the i-th output to attend to the first k input positions, after which the LogSparse attention is resumed, and (2) to divide the input sequence into subsequences, and to apply the LogSparse attention on each of them.

Qiu et al. [141] introduced BlockBERT, which relies on the block-wise attention: the input sequence is split into n_b non-overlapping blocks, and positions in block i are only allowed to attend to positions in block $\pi(i)$, where π denotes a permutation. The author chose to generate the permutations by simply shifting the positions. For instance, the possible permutations of $\{1, 2, 3\}$ are $\{1, 2, 3\}$, $\{3, 1, 2\}$, and $\{2, 3, 1\}$. The permutation $\{2, 3, 1\}$ means that the first block attends to the second block, the second block attends to the third block, and

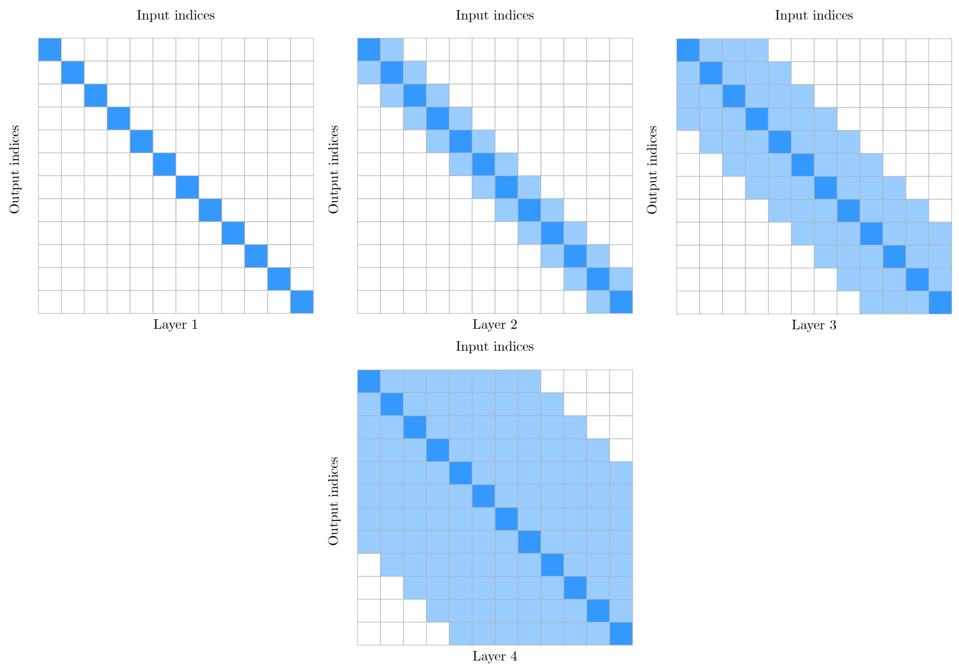


Figure 3.12 The connectivity matrices of the Cascade attention [139] for the first four layers with a base window $b = 1$ and a cardinal number $m = 2$. For instance, the window size of the third layer ($l = 2$) is equal to $2 \times b \times m^l - 1 = 7$.

the third block attends to the first block. In the multi-head setting, a different permutation[10] is assigned to each head. More formally, the output position i is only allowed to attend to input j if the following condition is satisfied:

$$\pi\left(\left\lfloor \frac{(i-1)n_b}{n} + 1 \right\rfloor\right) = \left\lfloor \frac{(j-1)n_b}{n} + 1 \right\rfloor \tag{3.14}$$

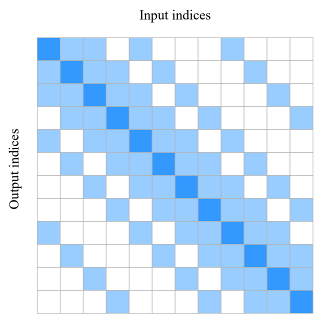


Figure 3.13 The connectivity matrix of the LogSparse attention Li et al. [140].

[10]Note that if the number of heads is greater than the number of permutations, multiple heads must be assigned the same permutation.

Figure 3.14 illustrates the connectivity matrix of the block-wise attention where a sequence of length $n = 12$ is split into $n_b = 3$ blocks. Although the block-wise attention reduces the memory and computational cost by a factor n_b, the complexity remains quadratic with respect to the sequence length.

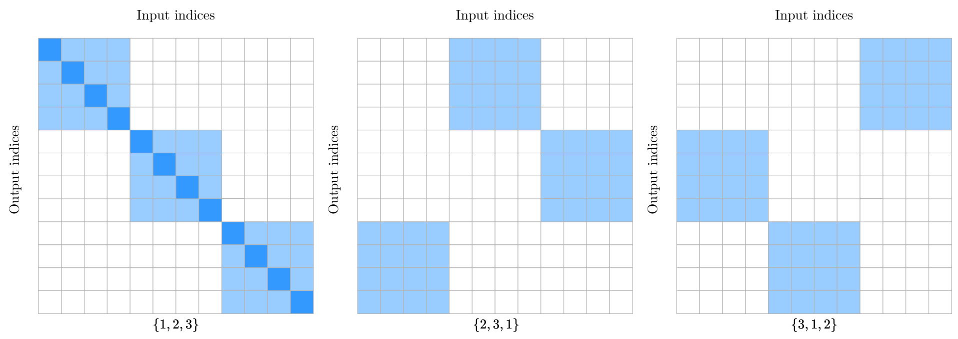


Figure 3.14 The connectivity matrices of the block-wise attention [141] for $n_b = 3$ blocks. The corresponding permutations are written below the connectivity matrices.

Beltagy et al. [142] introduced the Longformer which further reduces the complexity to $\mathcal{O}(n)$ using a combination of sliding window and global attentions (see Figure 3.15). The assumption behind the sliding window attention is that the most useful information is located in each position's neighbourhood. The sliding window attention is limited in that it requires $\mathcal{O}(\sqrt{n})$ layers to model long-range dependencies. Thus, a few preselected tokens have a global attention: they can attend to every position and be attended by every position. Consequently, the maximum path length between any two positions is equal to 2. Zaheer et al. [58] introduced BigBird, which also achieves a linear complexity using a combination of random, sliding window, and global attentions (see Figure 3.15). BigBird has two configurations that the authors referred to as internal transformer construction (ITC) and extended transformer construction (ETC). Similarly to the Longformer, the former uses existing positions for global attention, while the latter uses additional tokens, increasing the model's capacity and performance. Interestingly, the extra location of ETC may be seen as a form of memory. The authors proved that their sparse factorization preserves the theoretical properties of Transformers with the full attention: the model is both a universal approximator of sequence functions and Turing complete. However, BigBird without random attention outperformed BigBird with it in most of their experiments.

Learned and Adaptive Sparse Patterns [143, 144, 145]: Fixed and random patterns are handcrafted and may not be suitable for the data and task at hand. One may instead learn the relevant patterns and adapt them based on the content.

In order to increase the flexibility of the block-wise attention, Tay et al. [143] introduced the

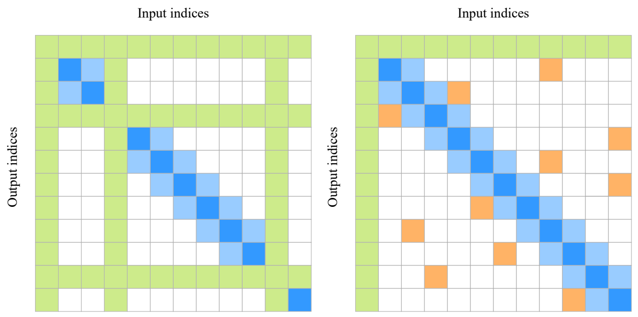


Figure 3.15 The connectivity matrices of two sparse attention schemes. (Left) Longformer [142]. (Right) BigBird [58]. The attention is the combination of sliding window attention (blue), global attention (green), and random attention (orange).

sparse Sinkhorn attention, which is equivalent to the block-wise attention whose keys have been sorted in a block-wise fashion. In other words, the permutations are learned. More specifically, the sparse Sinkhorn attention transforms the input sequence $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ into $\boldsymbol{X}' \in \mathbb{R}^{n_b \times d}$ where n_b is the number of blocks, and where $\boldsymbol{X}'_i$ is equal to the sum of the input in that block. A simple feed-forward network then learns a mapping $\boldsymbol{R}_i \in \mathbb{R}^{n_b}$ from the i-th block $\boldsymbol{X}'_i$ to all blocks. In order to obtain a sorting matrix from $\boldsymbol{R} \in \mathbb{R}^{n_b \times n_b}$, that is, a matrix comprising only 0s and 1s, and whose rows and column sum to one, the rows and columns are iteratively normalized. The sorting matrix is then used to permute the keys, effectively learning which block to attend (see Figure 3.16). The sparse Sinkhorn attention reduces the complexity to $\mathcal{O}(n_b^2)$. Nonetheless, since the block size is constant in the original paper, the complexity remains quadratic with respect to the sequence length. Additionally, the authors proposed a truncated version of the sparse Sinkhorn attention, which selects a few keys after sorting them, further reducing the complexity to $\mathcal{O}(n)$.

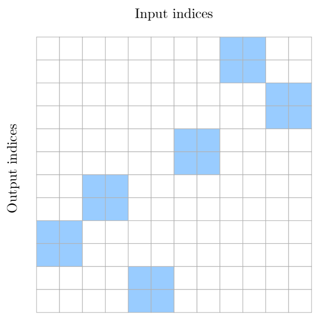


Figure 3.16 The connectivity matrix of the sparse Sinkorn attention [143].

Recently, Shi et al. [144] put under the microscope the attention patterns learned by BERT [51] and observed that the diagonal elements are less important compared to other positions, that is, they contribute the least to the output, while neighbourhood positions and special tokens are prominent. To confirm their observations, they dropped the diagonal element in BERT's attention such that each position is not allowed to attend to itself and noted that the performance remains comparable to the original model. Additionally, they observed that models for different tasks have various degrees of redundancy and hence can achieve various sparsity levels before significantly dropping performance. Consequently, Shi et al. [144] proposed to learn sparsity patterns for each task in an end-to-end fashion with the Differentiable Attention Mask (DAM) algorithm. Let us denote the attention score between the i-th output position (query) and j-th input position (key) as $\alpha_{i,j}$. They proposed to compute the attention mask $M_{i,j}$ as the Gumbel-Sigmoid [146] of the attention score $\alpha_{i,j}$:

$$M_{i,j} = \text{Gumbel-Sigmoid}(\alpha_{i,j}) = \text{Sigmoid}\left(\frac{\alpha_{i,j} + G_1 - G_2}{\tau}\right) \tag{3.15}$$

where G_1, G_2 are independent Gumbel noises $G_k = -\log(-\log(U_k))$ generated from a uniform distribution $U_k \sim \mathcal{U}(0, 1)$, and where τ is a temperature hyperparameter. Note that the Gumbel-Sigmoid becomes binary as τ approaches 0. A penalty term $\lambda \|M\|_1$ is added to the loss to control the trade-off between performance and sparsity. The resulting model called SparseBERT achieved 91.2% sparsity while maintaining an average score of 80.9% on GLUE, i.e., only 3% lower than the full BERT. Such an approach deviates from previous sparse attention whose patterns have been manually handcrafted. To avoid learning completely unstructured sparsity patterns, the authors proposed to enforce the first and last row/column of the attention mask to be active and all positions on each line parallel to the diagonal to share their mask parameters.

As mentioned above, due to the exponential nature of the Softmax, most positions are lightly attended to. In other words, most attention weights are small but non-zero. Instead, Correia et al. [145] introduced the Adaptively Sparse Transformer that replaces the Softmax by the α-entmax function, a differentiable generalization of the Softmax that pushes small weights to be exactly zero. Formally, the α-entmax function is defined as:

$$\alpha\text{-entmax}(\boldsymbol{z}) = \underset{\boldsymbol{p} \in \Delta^d}{\text{argmax}}\ \boldsymbol{p}^\top \boldsymbol{z} + \boldsymbol{H}^T_\alpha(\boldsymbol{p}), \tag{3.16}$$

where $\Delta^d = \{\boldsymbol{p} \in \mathbb{R}^d : \sum_i p_i = 1\}$ and, for $\alpha \geq 1$, $\boldsymbol{H}^T_\alpha$ is the Tsallis continuous family of entropies:

$$\boldsymbol{H}^T_\alpha(\boldsymbol{p}) = \begin{cases} \frac{1}{\alpha(\alpha-1)} \sum_j (p_j - p_j^\alpha), & \alpha \neq 1 \\ -\sum_j p_j \log p_j, & \alpha = 1. \end{cases} \tag{3.17}$$

The authors showed that the solution to the equation 3.16 is

$$\alpha\text{-entmax}(\boldsymbol{z}) = [(\alpha-1)\boldsymbol{z} - \lambda\mathbf{1}]_+^{\frac{1}{\alpha-1}}, \tag{3.18}$$

where $[]_+$ denotes the ReLU function, $\mathbf{1}$ denotes the vector of ones, and λ is the Lagrange multiplier corresponding to the $\sum_i p_i = 1$ constraint.

Interestingly, when $\alpha = 1$, the α-entmax is equivalent to the Softmax, and the attention is dense, and when $\alpha > 1$, the output is permitted to be sparse. In their experiments, a scalar parameter $a_{i,j}$ is learned for the j-th attention head of the i-th layer, and $\alpha_{i,j}$ is computed as:

$$\alpha_{i,j} = 1 + \text{sigmoid}(a_{i,j}) \in \,]1, 2[\tag{3.19}$$

Nonetheless, the Adaptively Sparse Transformer computes the attention score for each pair of queries and keys. Consequently, the sparsity cannot be leveraged to improve the memory and computation, resulting in a model that is 25% slower than the original Transformer in terms of tokens per second.

As of this survey's writing, unstructured sparse attention (whether fixed, random or learned) does not benefit from efficient implementations and therefore cannot result in memory and computational improvements. Nonetheless, there are exciting researches in that direction, as noted by Hooker [147]. In contrast, some structured sparsity patterns benefit from efficient implementations. Recently, NVIDIA introduced its Ampere architecture which efficiently compresses 2:4 structured sparsity on rows, that is, two non-zero values in every four entries.

Clustering and Locality-Sensitive Hashing [87, 148]: The Softmax function is dominated by the largest values, that is, by the keys and queries that have the largest dot product. Therefore, the attention may be approximated by only comparing the most similar keys and queries. Although this approach is a form of adaptive sparsity as the patterns depend on the data, they are presented separately due to their conceptual difference.

Kitaev et al. [87] introduced the Reformer, which selects the set of keys that the query can attend to by grouping them with an angular multi-round locality-sensitive hashing (LSH). Such hashing scheme has a high probability of assigning the same value to similar vectors. Formally, queries and keys are shared ($Q = K$) and bucketed using b hash values obtained as follows:

$$\boldsymbol{p} = [\boldsymbol{x}^\top \boldsymbol{R}; -\boldsymbol{x}^\top \boldsymbol{R}] \tag{3.20}$$

$$h(\boldsymbol{x}) = \underset{i}{\text{argmax}}(p_i) \tag{3.21}$$

where ; denotes the concatenation operation, and where $\boldsymbol{x} \in \mathbb{R}^d$ is a query/key and $\boldsymbol{R} \in \mathbb{R}^{d \times b/2}$ is a random rotation matrix. Output positions are only allowed to attend to input positions that are in the same bucket. They are, however, not allowed to attend to themselves because the dot product of a vector with himself will almost always be greater than the dot product with other positions.

The authors chose a constant bucket size l_B, resulting in a number of buckets $n_B = n/l_B$. The attention complexity is $\mathcal{O}(n_B \times l_B^2)$ which simplifies as $\mathcal{O}(n)$. This does not take into account the computation of the hash values for each position. As only $\log n_B$ bits are required to encode n_B buckets, the complexity of computing hash values is given by $\mathcal{O}(n \log n_B)$, which simplifies as $\mathcal{O}(n \log n)$. Consequently, the complexity of the Reformer's attention is $\mathcal{O}(n \log n)$.

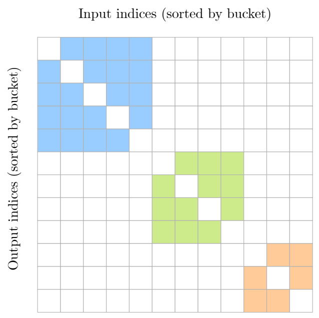

Input indices (sorted by bucket)
Output indices (sorted by bucket)

Figure 3.17 The connectivity matrix of the Reformer [87]. Queries and keys are bucketed using LSH then sorted by their bucket. Therefore, the i-th row of the connectivity matrix may not correspond to the i-th position in the input sequence. Units can only attend other units in the same bucket, but not themselves because queries and keys are equal. The colour represents buckets.

The Maximum Inner Product Search (MIPS) problem is the task of searching for the vector K_j in $K = \{K_1, K_2, \cdots, K_n\}$ that maximizes the dot product with a given vector Q_i. Note that the MIPS problem is particularly useful for the attention mechanism as $Q_i^\top K_j$ is directly proportional to the contribution of the j-th value for the i-th attention's output. There are multiple approaches to approximately solve this problem, including tree-based and LSH-based. When the norm of every K_j is constant, the problem is equivalent to the Nearest Neighbour Search (NNS). Motivated by this observation and to avoid the computational cost of learning sparsity patterns, Roy et al. [148] proposed the Routing Transformer that relies on an online mini-batch version of k-means and a set of centroids learned along the rest of the parameters. Like the Reformer, queries can only attend to keys from the same cluster, inducing an adaptive or content-based sparsity pattern.

3.4.2 Factorized Attention

Wang et al. [88] demonstrated that the attention matrix Softmax$\left(\boldsymbol{Q}\boldsymbol{K}^\top/\sqrt{d}\right)$ is approximately low rank. Consequently, another approach to reduce the Transformer's complexity is to approximate the attention by factorizing it into the product of two matrices with lower dimensions.

Low-Rank Factorization [88, 149, 150]: Wang et al. [88] introduced the Linformer, a linear complexity model that approximates the attention with a low-rank factorization by first projecting each key to a lower dimension before performing the dot product, thereby saving time and memory. Formally, the low-rank attention is given by:

$$\text{Attention}(\boldsymbol{X}) = \underbrace{\text{Softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}}\right)}_{n\times n}\underbrace{\boldsymbol{V}}_{n\times d} \approx \underbrace{\text{Softmax}\left(\frac{\boldsymbol{Q}(\boldsymbol{E}\boldsymbol{K})^\top}{\sqrt{d}}\right)}_{n\times k}\underbrace{\boldsymbol{F}\boldsymbol{V}}_{k\times d} \tag{3.22}$$

where $\boldsymbol{E}, \boldsymbol{F} \in \mathbb{R}^{k\times n}$, with $k \ll n$, are two linear projection matrices learned during training. The authors showed that $\boldsymbol{E}$ and $\boldsymbol{F}$ could be shared across heads and layers with virtually no performance penalty.

Tay et al. [149] introduced a family of models called Synthesizers that learn the compatibility scores without computing the pairwise dot products between the queries and keys. For instance, the Dense Synthesizer learns the compatibility scores with a simple position-wise feed-forward network that projects each of the n rows of $\boldsymbol{X}$ from $\mathbb{R}^{1\times d}$ to $\mathbb{R}^{1\times n}$:

$$\text{F}(\boldsymbol{X}_i) = \max(0, \boldsymbol{X}_i\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2 \tag{3.23}$$

where $\boldsymbol{W}_1 \in \mathbb{R}^{d\times d}$ and $\boldsymbol{W}_2 \in \mathbb{R}^{d\times n}$. Finally, the attention is given by:

$$\text{Attention}(\boldsymbol{X}) = \text{Softmax}(F(\boldsymbol{X}))G(\boldsymbol{X}) \tag{3.24}$$

where $G(\cdot) : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d}$ is a projection of the input akin to the values. In order to improve the efficiency, the authors proposed the Factorized Dense Synthesizer which first project the input $\boldsymbol{X}$ with two feed-forward networks:

$$\boldsymbol{A} = F_A(\boldsymbol{X}) \in \mathbb{R}^{n\times a} \quad \text{and} \quad \boldsymbol{B} = F_B(\boldsymbol{X}) \in \mathbb{R}^{n\times b}, \tag{3.25}$$

such that $a \times b = n$. Then, two tiling functions $H_A(\cdot) : \mathbb{R}^{n\times a} \to \mathbb{R}^{n\times(a.b)}$ and $H_B(\cdot) : \mathbb{R}^{n\times b} \to \mathbb{R}^{n\times(b.a)}$ are applied to $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively. Note that a tiling function simply repeats a vector multiple times. Finally, the attention of the Factorized Dense Synthesizer is given by:

$$\text{Attention}(\boldsymbol{X}) = \text{Softmax}(H_A(\boldsymbol{A})H_B(\boldsymbol{B})^\top)G(\boldsymbol{X}) \tag{3.26}$$

Additionally, the authors proposed a baseline called the Factorized Random Synthesizer, whose compatibility scores are independent of the input. Formally, the Factorized Random Synthesizer's attention is given by:

$$\text{Attention}(\boldsymbol{X}) = \text{Softmax}(\boldsymbol{R}_1\boldsymbol{R}_2^\top)G(\boldsymbol{X}) \tag{3.27}$$

where $\boldsymbol{R}_1, \boldsymbol{R}_2 \in \mathbb{R}^{n\times k}$ are two low-rank matrices learned during training. Although the Synthesizers eliminate the need to compute the pairwise dot products, which speed up the model in practice, the complexity remains quadratic with respect to the sequence length.

The Nyströmformer [150] relies on the Nyström method to generate a low-rank approximation of the Softmax matrix. However, applying the Nyström method directly to the Softmax would require to compute the $\boldsymbol{Q}\boldsymbol{K}^\top$ product, which requires $\mathcal{O}(n^2)$ computations and memory. As a solution, the Nyströmformer creates two subsets $\tilde{\boldsymbol{K}}$ and $\tilde{\boldsymbol{Q}}$ of columns, called landmarks, from $\boldsymbol{K}$ and $\boldsymbol{Q}$, respectively. The authors applied the segment-means approach, which computes the landmarks as the averages over predefined spans of keys and queries. Let $\boldsymbol{S}_{AB}$ denotes $\text{Softmax}(\boldsymbol{A}\boldsymbol{B}^\top/\sqrt{d})$ for any matrix $\boldsymbol{A}$ and $\boldsymbol{B}$. The Nyströmformer approximates the Softmax matrix as:

$$\text{Softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}}\right) \approx \boldsymbol{S}_{Q\tilde{K}}\boldsymbol{S}^+_{\tilde{Q}\tilde{K}}\boldsymbol{S}_{\tilde{Q}K} \tag{3.28}$$

where the superscript + denotes the Moore-Penrose inverse typically computed with the singular value decomposition (SVD). Since the SVD is inefficient on GPU, the authors relied on an iterative method that approximate $\boldsymbol{S}^+_{\tilde{Q}\tilde{K}}$ as Z^+. Finally, the Nyströmformer's attention is given by:

$$\text{Attention}(\boldsymbol{X}) \approx \boldsymbol{S}_{Q\tilde{K}}Z^+\boldsymbol{S}_{\tilde{Q}K}\boldsymbol{V} \tag{3.29}$$

which can be efficiently encoded in a computational graph.

Provided that the number of landmarks is constant and much smaller than the sequence length, the Nyströmformer complexity is $\mathcal{O}(n)$. Depending on the number of landmarks and the sequence length, the authors reported substantial gains over the Linformer and Longformer on the masked language model and sentence order prediction objectives. Additionally, the representations learned by the Nyströmformer appear to transfer as well as BERT to different NLP tasks. Nonetheless, a more extensive evaluation of the Nyströmformer remains necessary.

Kernel Attention [151, 152]: A kernel $K(\cdot,\cdot)$ is a function that takes two vectors as arguments and returns the product of their projection by a feature map $\phi(\cdot)$:

$$K(\boldsymbol{x},\boldsymbol{y}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{y}) \tag{3.30}$$

Katharopoulos et al. [152] interpreted the Softmax as a kernel, decomposed it as an inner product in the right space, and rearrange the computations in a clever way to reduce the complexity. More specifically, the self-attention of a given query $\boldsymbol{Q}_i$ may be rewritten using a mapping $\phi(\cdot)$:

$$\text{Softmax}\Big(\boldsymbol{Q}_i^\top \boldsymbol{K}^\top\Big)\boldsymbol{V} = \frac{\sum_{j=1}^n \exp\Big(\boldsymbol{Q}_i^\top \boldsymbol{K}_j\Big)\boldsymbol{V}_j}{\sum_{j=1}^n \exp\Big(\boldsymbol{Q}_i^\top \boldsymbol{K}_j\Big)} = \frac{\sum_{j=1}^n \phi\Big(\boldsymbol{Q}_i\Big)^\top \phi\Big(\boldsymbol{K}_j\Big)\boldsymbol{V}_j}{\sum_{j=1}^n \phi\Big(\boldsymbol{Q}_i\Big)^\top \phi\Big(\boldsymbol{K}_j\Big)} = \frac{\phi\Big(\boldsymbol{Q}_i\Big)^\top \sum_{j=1}^n \phi\Big(\boldsymbol{K}_j\Big)\boldsymbol{V}_j^\top}{\phi\Big(\boldsymbol{Q}_i\Big)^\top \sum_{j=1}^n \phi\Big(\boldsymbol{K}_j\Big)} \tag{3.31}$$

where the scaling factor $\sqrt{d}$ has been omitted for the sake of readability. The authors noted that $\sum_{j=1}^n \phi\Big(\boldsymbol{K}_j\Big)\boldsymbol{V}_j^\top$ and $\sum_{j=1}^n \phi\Big(\boldsymbol{K}_j\Big)$ must only be computed a single time, therefore reducing the complexity from quadratic to linear both in terms of memory and computation. The vectorized formulation of the numerator makes it simpler to see:

$$\underbrace{\phi\Big(\boldsymbol{Q}\Big)}_{n\times p}\Big(\underbrace{\phi\Big(\boldsymbol{K}\Big)^\top}_{p\times n}\underbrace{\boldsymbol{V}}_{n\times d}\Big) \tag{3.32}$$

where the mapping $\phi(\cdot) : \mathbb{R}^d \to \mathbb{R}^p$ is applied position-wise. Unfortunately, the feature map of the exponential kernel is infinite dimensional. Hence, any finite kernel is an approximation of the attention matrix and may be interpreted as a low-rank factorization. However, they are presented separately here due to their conceptual difference. Katharopoulos et al. [152] approximated the attention matrix in the Linear Transformer with the feature map $\phi(x) = \text{elu}(x) + 1$, where the function $\text{elu}(\cdot)$ denotes the exponential linear unit given by:

$$\text{elu}(x) = \begin{cases} \alpha(e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases} \tag{3.33}$$

where α is an hyperparameter. The Linear Transformer performed on par with the vanilla Transformer on autoregressive image generation, but poorly on automatic speech recognition.

Choromanski et al. [151] later demonstrated that the exponential is equivalent to a kernel with a randomized mapping:

$$\exp(x^\top y) = \mathbb{E}_{w\sim\mathcal{N}(0,I_d)}\left[\exp\left(w^\top x \frac{\|x\|^2}{2}\right)\exp\left(w^\top y \frac{\|y\|^2}{2}\right)\right] \tag{3.34}$$

Consequently, the authors introduced the Performer, a linear complexity model that approximates the attention by means of a kernel with the following feature mapping:

$$\phi(x) = \frac{\exp(-\|x\|^2/2)}{\sqrt{2p}}\Big[\exp\Big(w_1^\top x\Big);\ldots;\exp\Big(w_p^\top x\Big);\exp\Big(-w_1^\top x\Big);\ldots;\exp\Big(-w_p^\top x\Big)\Big] \quad (3.35)$$

where $w_i \sim \mathcal{N}(0, I_d)$. To further reduce the variance of the estimator, w_i are constrained to be exactly orthogonal, which is achieved with the Gram-Schmidt process. The hyperparameter p corresponds to the number of random features and controls the quality of the approximation.

Clustering and Locality-Sensitive Hashing [153]: As previously explained, clustering can uncover sparse patterns by grouping queries and keys and only computing the attention between positions within the same cluster. Alternatively, Vyas et al. [153] proposed to factorize the attention with clustering by grouping queries into a fixed number of non-overlapping clusters and by computing the attention between the cluster's centroids and the keys. Consequently, the attention score is only computed once per group of similar queries and broadcasted to all, resulting in linear complexity. Since queries may be clustered differently across attention heads and since the attention sub-layer includes a residual connection, two queries in the same cluster can have different output representations. The authors proved that the approximation error for a given query is bounded by its distance to its centroid multiplied by the spectral norm of the keys matrix. As such, the K-Means algorithm can be used for minimizing the approximation error. However, K-Means in the original space would be slow to compute as Lloyd algorithm has a complexity of $\mathcal{O}(ncdl)$, where c is the number of clusters and l is the number of Lloyd iterations. Instead, the authors first used a locality-sensitive hashing scheme on the queries before applying K-Means with the Hamming distance, which reduces the complexity to $\mathcal{O}(ncl + cbl + ndb)$, where b is the number of bits used for hashing.

To further improve the approximation, Vyas et al. [153] proposed the *improved cluster attention* that separately consider the k keys with the highest attention for each cluster. Intuitively, keys with high approximated attention may have low attention for some queries, resulting in a large approximation error. As a solution, the dot product between these top-k keys and all queries belonging to the corresponding cluster is computed. Then, the attention is rescaled by the total probability mass assigned to these top-k keys.

Compared to the Reformer, Vyas et al. [153] method is significantly faster (43% lower epoch time) while being significantly more accurate (35% lower phone error rate) for speech recognition on the Wall Street Journal.

3.4.3 Architectural Change

Finally, the Transformer's complexity may also be reduced by modifying the model's architecture and preserving the original attention mechanism. Let us investigate (i) the Transformer-XL and the Compressive Transformer that rely on memory, and (ii) then the Funnel-Transformer that iteratively compresses sequences.

Memory [48, 154]: The block-wise approach splits the input sequence into small non-overlapping subsequences called windows, blocks, or chunks, which are processed independently; therefore, the maximum dependency length is equal to that of the subsequence. To leverage information from previous windows, Dai et al. [48] introduced the Transformer-XL, which relies on segment-based recurrence between windows. This recurrence mechanism is implemented by storing the representations of the previous window in a first-in first-out memory (FIFO). Then, the attention mechanism can attend to the representations located in this memory, but the gradients are not computed for the attention on these elements. Although this model achieves a RECL four times greater than the vanilla Transformer with the same parameter budget, it cannot capture dependencies outside the FIFO memory range. Furthermore, this model is only compatible with autoregressive tasks. This technique is analogous to truncated backpropagation through time (BPTT), except that a sequence of hidden states is considered instead of the previous one. Figure 3.18 illustrates the segment-based recurrence of the Transformer-XL.

In order to further increase the range of dependencies considered by the Transformer-XL, Rae et al. [154] proposed the Compressive Transformer, which adds a compressed memory to the original FIFO memory. Representations of past windows are first stored in the standard FIFO memory, like the Transformer-XL. Then, when this memory is full, the oldest representations are compressed with a user-defined function and stored in the compressed FIFO memory instead of being discarded. The number of elements considered in the original FIFO memory to generate the compressed memory depends on the chosen function. The authors propose using max/mean pooling, 1D convolution, dilated convolutions, or the most attended representations by the attention. They also proposed to learn the compression function with an auxiliary auto-encoding loss and a variant called attention-reconstruction loss, which typically reconstructs the original memory from the compressed ones. They show a clear advantage over the Transformer-XL on NLP tasks and comparable results on speech modelling.

Sequence Compression [155]: Many tasks such as image classification and sentiment analysis only require producing a single output for the whole sequence. Dai et al. [155] argued that the full-length sequence of hidden states may contain significant redundancy and that

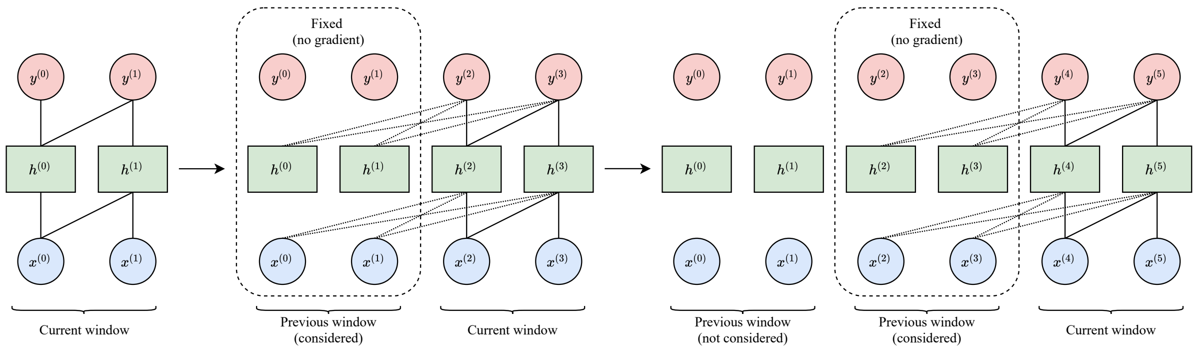


Figure 3.18 Segment-based recurrence, which is similar to truncated BPTT. The window size is equal to two, and only the previous window is considered. For the sake of clarity, parameters from and to states that do not contribute are omitted.

the model may not have to preserve token-level information. Consequently, they proposed the Funnel-Transformer, whose encoder reduces the computational cost by gradually reducing the length of the hidden states sequence with pooling. Note that instead of directly feeding the pooled sequence into the attention layer, it is only used to construct the query matrix, while the unpooled sequence is used to construct the key and value matrices. Additionally, the authors proposed to recover the original sequence length by up-sampling the compressed sequence of hidden states to address the common pre-training objectives, such as MLM, that require separate representation for each token. Although the Funnel-Transformer effectively reduces the computational and memory cost of the encoder, the complexity remains quadratic, and the best performances are achieved on tasks that only require sequence-level representation.

3.5 Shortcomings

This section discusses the lack of understanding of the self-attention inner workings and the limitation of the Transformer evaluation methodology, including the lack of standard benchmarks for long-range dependencies.

Self-attention is a relatively new mechanism that has been quickly and widely adopted due to its remarkable empirical success. Nonetheless, the self-attention inner workings are not yet fully understood, and many questions remain unanswered, including why it works, what it learns, and whether it is interpretable. Answering those questions is crucial to designing faster and lighter Transformers that are competitive with the original model. As of this paper's writing, the deep learning community actively investigates self-attention and have proposed preliminary answers to the aforementioned questions. For instance, evidence supporting both the self-attention interpretability [156, 157] and non-interpretability [158] have been published. Tay et al. [149] empirically evaluated the dot product impact on natural language processing tasks and concluded that query-keys interaction is "*useful but not that important*". Kitaev et al. [87] investigated the impact of sharing queries and keys, and concluded that "*it turns out that sharing QK does not affect the performance of Transformer*".

Despite our current limited understanding of the self-attention mechanism, a wide range of faster and lighter Transformers have been introduced in a short amount of time, each claiming comparable or superior performance to the vanilla Transformer. Since there is no consensus on how to evaluate the proposed approaches [159], researchers often have to evaluate their method on a small range of tasks. However, different tasks may require different assumptions, which means that one method may work well on a specific task but poorly on others. For instance, Tay et al. [149] showed that a simple Synthesizer is highly competitive with the vanilla Transformer across a range of natural language processing tasks, including machine translation, language modelling, and text generation. However, Tay et al. [159] later showed that the vanilla Transformer outperforms the Synthesizer on the more difficult Long-Range Arena benchmark. Long-Range Arena [159] is a suite of five general and challenging tasks designed to evaluate how well Transformers capture long-term dependencies from different modalities such as text, natural and synthetic images, and mathematical expressions. Table 3.3 compiles the Long-Range Arena results of the models discussed in the survey. For a complete description of the objectives and datasets, we refer the reader to the original paper.

Furthermore, due to Transformers large training cost, researchers often evaluate their approach against a limited number of models on the tasks of interest. For instance, [87] only evaluated the Reformer against three distinct vanilla Transformers [43, 160] on three tasks. Standardized suites of benchmarks such as GLUE and the recent Long-Range Arena allow researchers and practitioners to evaluate only their method and compare it against a public leaderboard. Consequently, we highly recommend that researchers consider such benchmarks.

Although standardized benchmarks such as Long-Range Arena would help compare the models, the results should be taken with caution since the performance depends on the model size and hyperparameters, the speed depends on the implementation and hardware, and the memory footprint depends on the implementation and general methods used. For instance, the Switch Transformer uses a mixture of experts, mixed-precision, expert dropout, knowledge distillation, and a careful initialization. Therefore, it is difficult to isolate the benefit of a single modification.

Finally, the complexity is not always representative of the practical efficiency. For instance, the Reformer achieves an asymptotic complexity of $\mathcal{O}(n \log n)$ but is significantly slower than the vanilla Transformer on small sequences, as shown in Table 3.3. This slow down is due to large constants hidden in the complexity. Even when there are no hidden constants, there is a distinction between theoretical complexity and what is achievable in practice. For instance, sparse matrix multiplication may reduce the complexity from quadratic to linear in theory. However, it is well known that GPUs and TPUs are not designed to perform such operations efficiently [161] and, in practice, sparse matrix multiplication is often slower than dense ones. We encourage researchers to explicitly report the complexity as well as the number of floating operations (FLOPs), the wall-clock time with the hardware, and the memory footprint of their method.

Table 3.3 Long-Range Arena benchmark [159]. Results have been compiled from the original paper. Benchmarks are run on 4x4 TPU V3 chips, and the memory is reported per device.

Models	Average score (%)	Steps per second		Peak memory (GB)	
		1K	4K	1K	4K
Transformer [43]	54.39	8.1	1.4	0.85	9.48
Sparse Transformer[10] [138]	51.24				
Longformer[10] [142]	53.46				
BigBird [58]	**55.01**	7.4	1.5	0.77	2.88
Sinkhorn Transformer [143]	51.39	9.1	5.3	0.47	1.48
Reformer [87]	50.67	4.4	1.1	0.48	2.28
Linformer [88]	51.36	9.3	7.7	**0.37**	**0.99**
Synthesizer [149]	51.39	8.7	1.9	0.65	6.99
Linear Transformer [152]	50.55	9.1	7.8	**0.37**	1.03
Performer [151]	51.41	**9.5**	**8.0**	**0.37**	1.06

3.6 Broader Impact of Efficient Transformer

This section extends the three motivations and potential impacts of lighter and faster Transformers briefly discussed in Section 3.2.4.

First and foremost, computational resources are not only finite but also expensive. Consequently, there are severe inequalities between research groups and between companies. Indeed, many researchers do not have access to GPU or TPU farms, and most companies cannot afford to spend thousands or millions of dollars on dedicated hardware, especially if deep learning is not their primary focus. At this time, the resources disparities have increased dramatically to a point where only a few parties can afford to train massive state-of-the-art

[10]The Sparse Transformer and Longformer depends on CUDA kernels that are difficult to implement on TPUs. Therefore, Tay et al. [159] used *equivalent* implementations to emulate their performance and did not report their efficiency.

models. A prime example of this cleavage is the Transformer. Indeed, the largest Transformers are so expensive to train, even for large companies such as Microsoft, that they are only trained once. For instance, Brown et al. [50] noticed an issue in their pre-processing after training GPT-3. As the author explained, they could not train their model again due to the massive cost and therefore published their results with a known issue. Resources inequalities also hinder creativity as researchers with promising ideas may not be able to implement them, thus reinforcing the vicious "rich get richer" circle, where well-funded groups and companies that have access to more resources are more likely to achieve state-of-the-art results and receive more fundings [79].

Additionally, lower-complexity Transformers enable novel applications as extremely long sequences cannot be processed in a reasonable amount of time by the quadratic complexity vanilla Transformer. For instance, Choromanski et al. [151] observed the Performer's potential impact on biology, and Zaheer et al. [58] evaluated BigBird on genomics tasks that take fragments of DNA as input. Huang et al. [162] were able to generate minute-long musical compositions with a Transformer that leverage the block-wise approach and an efficient computation of the relative attention. Note that contrary to the attention introduced by [43], the relative attention [163] explicitly models the input positions. The range of applications will surely expand as researchers design ever-lighter and -faster Transformers.

Finally, recent research made it clear that we must cut carbon dioxide (CO2) emissions in half over the next decade to limit global warming. The large-scale infrastructures used by the deep learning community consume a considerable amount of electricity, which is mainly produced by non-renewable sources such as coal or gas [80]. Strubell et al. [79] estimated that training a Transformer with neural architecture search generates up to 284,000 kg of CO2. For reference, the average American emits 16,400 kg of CO2 per year, and the average car emits about 57,200 kg during its lifetime[11] (fuel included). The authors estimated that training a single instance of BERT [51] on GPU produces about the same amount of CO2 as a trans-American flight. Although lighter and faster models require fewer resources and therefore produce less carbon dioxide, they are also more accessible, so we would expect more models to be trained. Overall, it is difficult to know whether lighter and faster Transformers will positively impact the environment. Nonetheless, researchers and practitioners ought to have in mind the significant environmental impact of their experiments, which can be estimated with the Machine Learning Emissions Calculator[12] developed by Luccioni et al. [164].

[11]A product lifetime or lifecycle typically includes material production, manufacturing, usage, and end-of-life disposal.

[12]`https://mlco2.github.io/impact/`

3.7 Future Research Directions

In our opinion, the current research directions follow one of two purposes: (i) efficiency and affordability or (ii) generalization performance. Since this survey addresses approaches to yield faster and lighter Transformers, let us start with the efficiency and affordability objective.

3.7.1 Efficiency and Affordability

To the best of our knowledge, researchers and practitioners have not yet identified a specialized approach that improves the Transformer's efficiency for every task, dataset, and hardware, as explained in Section 3.5. In our opinion, one of the most promising avenues is to learn adaptively sparse patterns that are structured for the available hardware. Let us justify our claim.

The Softmax function only contains a few large values due to its exponential nature. Therefore, it can be effectively approximated by masking the positions with small weights. In theory, the computation and memory reduction is linearly proportional to the ratio of masked positions. In practice, however, the improvement depends on the hardware. As of this survey's writing, NVIDIA is the first and only manufacturer to offer an architecture that natively supports sparse operations, resulting in a virtually perfect speed-up. One may reasonably expect other manufacturers to follow this direction due to the prevalence of sparse operations in deep learning. Therefore, the sparse patterns should be structured such that the hardware natively supports them. Handcrafting features or patterns based on prior knowledge is known to be suboptimal. Instead, the model should learn the patterns from the data for the task at hand. Additionally, individual samples are likely to require different attention patterns, and hence, the patterns should be adaptative (content-based). Finally, we believe it is beneficial to include global tokens since they allow any position to attend to any other position in two layers, thus preserving the attention's expressiveness.

3.7.2 Generalization Performance

A second research venue consists in improving the network generalization performance. Since the deep learning renaissance associated with greedy layer-wise unsupervised pretraining [28], there has been a clear trend towards scaling up neural networks. As a result, researchers and practitioners have been able to leverage ever-larger datasets and ultimately improve the network's performance. In this setting, scaling is performed typically by increasing the number of layers, the number of attention heads, the input embedding dimension, and the feedforward network width.

Amongst others, Radford et al. [165] introduced a large Transformer called GPT-2 and evaluated various model sizes on language modelling tasks in a zero-shot setting. The authors reported that the performance significantly increased with the model size ranging from 117M to 1.5B parameters. Recently, Brown et al. [50] introduced GPT-3 based on the GPT-2 architecture and considered an even wider span of model sizes, ranging from 125M to 175B parameters. The authors reported that the model performance smoothly increased with the model size in most cases and suggested that this trend should extend to even larger models. Furthermore, Devlin et al. [51] investigated the effect of BERT size on the GLUE benchmark and concluded that "*larger models lead to a strict accuracy improvement across all four datasets, even for MRPC which only has 3,600 labeled training examples, and is substantially different from the pre-training tasks*".

These observations suggest that researchers and practitioners must scale their model to pursue the generalization performance objective. Inherently, scaling is resource-expensive and goes against the affordability sought in this survey. Nonetheless, there are research directions to improve the generalization capability of deep learning models that are orthogonal to scaling and thus compatible with efficiency. A promising avenue is structural inductive biases. A recent structural inductive bias inspired by independent mechanisms in the causality literature consists of designing an architecture that learns sparsely interacting modules, each one of them specialized in a different mechanism [166]. Ideally, individual modules should be robust to changes in the aspects of the world that are unrelated to this module, such as in the case of distributional shift. Lamb et al. [167] applied this idea to Transformers by introducing the Transformers with Independent Mechanisms (TIM). The authors observed that TIM layers could be combined with the mixture of experts approach, allowing the switching to be specific to distinct aspects of the data.

Combining universally effective and efficient approaches such as the aforementioned sparse patterns with conditional computing and the independent mechanisms prior appears to be promising to tackle complex tasks without relying on large-scale resources.

3.8 Conclusion

Transformers have quickly become the de facto model for processing sequences, notably achieving state-of-the-art in most natural language processing tasks at the cost of quadratic complexity. As a result, researchers have leveraged numerous techniques to mitigate this memory and computational burden. This survey investigated popular general methods to make neural networks lighter and faster and discussed their strengths and limitations. Notably, we advised researchers and practitioners to use mixed-precision and gradient checkpointing due to their simplicity and overall benefits. Often, these general techniques are not sufficient to mitigate the Transformer's complexity. Consequently, this survey reviewed the lower-complexity variations of the Transformer and discussed their assumptions, justification and shortcomings. Notably, we advised researchers and practitioners to rely on pre-trained models whenever possible. Otherwise, we recommend training a small vanilla Transformer with mixed-precision and gradient checkpointing to apprehend the dependencies required for the task and select the appropriate models accordingly. Additionally, we discussed the potential impacts of affordable Transformers, including improving the state-of-the-art, extending the range of applications, increasing the equity between researchers, and potentially reducing the environmental impact. Finally, we highlighted promising future research directions for this exciting architecture.

3.9 Acknowledgment

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena, and EffciOS for funding this research.

CHAPTER 4 ORGANIZATION OF THE THESIS

In addition to the aforementioned survey, this thesis is composed of three articles presented in chronological order of their submission in Chapters 5, 6, and 7. These chapters address each one of the research objectives defined in Chapter 1. Let us briefly describe how the three chapters fit together.

In Chapter 5, the scope is first reduced to performance anomalies, also referred to as latency, in order to become familiar with the task and subject at hand. This decision is motivated by the interest of the partnering company as well as the interests of the community, as indicated by the high number of publications on this subject. The first contribution of this thesis follows the literature while remaining simple: the proposed approach relies on popular off-the-shelf machine learning techniques and hand-crafted features. Albeit simple, the approach is able to effectively detect latency and provide insight into their potential underlying root cause. Most notably, the approach identified a genuine PHP cache contention issue responsible for real-world latency. Nonetheless, hand-crafted features are often specific to the data and task considered while being suboptimal, time-consuming, and error-prone.

In Chapter 6, the aforementioned limitations of the hand-crafted features are addressed by proposing a method to automatically learn a representation of the system calls along with their arguments. Instead of addressing the anomaly or novelty detection task, the second contribution of this thesis focuses exclusively on proposing a representation method and investigating which aspects of the system calls are instrumental in improving the performance of neural networks. Nonetheless, in anticipation of the third contribution of this thesis, the proposed representation technique has been evaluated on two natural language processing tasks with two popular neural networks that could subsequently be used to detect novelties. Furthermore, a modern and large dataset of web requests has been collected to evaluate the proposed approach and released for researchers to explore.

In Chapter 7, the scope is broadened to the detection of novelties defined as any deviation from previously observed behaviors. The third contribution of this thesis leverages the afore representation, as well as three prevalent neural networks for sequence processing and the left-to-right language modeling task, whose objective is to predict each token knowing the previous ones. The left-to-right LM allows computing the likelihood of sequences, thereby detecting novelties based on the idea that they would have a low likelihood under the model since they deviate from previously observed behaviors by definition. Notably, the proposed methodology is evaluated on the Transformer to investigate whether the ability to learn extremely long-term dependencies is beneficial and a lighter model that would enable scaling the approach to longer sequences. Furthermore, the third contribution of this thesis also extends on the second one regarding the dataset by releasing a larger dataset of web requests that comprises multiple behaviors, which is required to evaluate the novelty detection.

CHAPTER 5 ARTICLE 2: AUTOMATIC CAUSE DETECTION OF PERFORMANCE PROBLEMS IN WEB APPLICATIONS

Authors Quentin Fournier, Naser Ezzati-jivan, Daniel Aloise, and Michel R. Dagenais.

Published at 3rd International Workshop on Software Faults (30 October 2019)

5.1 Preface

As the two papers included in Chapters 5 and 6 need to be more explicit about how the traces were collected and subsequently split into sets, let us clarify these essential steps here. Please note that this section is not part of the following paper.

The first paper relies on DBSCAN and k-means, two clustering algorithms. Such unsupervised methods do not require splitting the data into multiple sets. Consequently, a single trace was collected at runtime, and the algorithms were applied to all the requests.

The second paper relies on neural networks to learn a language model. Such self-supervised methods require splitting the data into three sets: (1) a training set to learn the parameters, (2) a validation set to estimate the generalization, allowing for the hyperparameters to be updated accordingly, and (3) a test set held out for the final evaluation. Consequently, two traces were collected one after the other in order to prevent any unexpected overlap. Furthermore, the server was restarted to reset the process and thread ids. The first trace was entirely used for training, while the second trace was randomly split into a validation set (25%) and a test set (75%). Importantly, there is no overlap between the two evaluation sets, and no extensive hyperparameter search was conducted, which could have resulted in an adaptation to the validation set. The same approach was applied to obtain Ciena's datasets.

Abstract The execution of similar units can be compared by their internal behaviors to determine the causes of their potential performance issues. For instance, by examining the internal behaviors of different fast or slow web requests more closely, and by clustering and comparing their internal executions, one can determine what causes some requests to run slowly or behave in unexpected ways. In this paper, we propose a method of extracting the internal behavior of web requests as well as introduce a pipeline that detects performance issues in web requests and provides insights into their root causes. First, low-level and fine-grained information regarding each request is gathered by tracing both the user space and the kernel space. Second, further information is extracted and fed into an outlier detector. Finally, these outliers are then clustered by their behavior, and each group is analyzed separately. Experiments revealed that this pipeline is indeed able to detect slow web requests and provide additional insights into their true root causes. Notably, we were able to identify a real PHP cache contention issue using the proposed approach.

Keywords Performance Analysis, Cause Analysis, Tracing, Machine Learning, Clustering, Web Application.

5.2 Introduction

While it is crucial to identify performance issues, they can also be extremely difficult to detect [168] as they are rare and hard to reproduce. Additionally, detection techniques usually require specifying an unknown normal behavior.

Once a performance issue has been detected, developers often use debuggers and profilers to analyze the issue and identify its root cause. However, debuggers are rarely applicable as they operate by stopping the world — the program — which may, in fact, mask problems related to latency or race conditions. Profilers are also ineffective as they operate by averaging metrics, which hides outliers.

Execution tracing overcomes the drawbacks of debuggers and profilers by instrumenting the system at different levels and by collecting information at run time. It works by executing a macro that generates an event for each instance of tracepoints met during execution. Tracing permits low-level and fine-grained information to remain on the system, which aids in discovering the root cause of the problem. However, the size of the collected trace may be enormous in complex, modern systems.

As the size and complexity of trace data continue to increase, the need for automated analysis becomes equally as important. However, automating this process is complex. A system may perform slowly for numerous reasons: an improper configuration, a change in the environment, unusual and possibly malicious network traffic, a software bug, or simply an inefficient code modification. An expert is only able to dissociate the aforementioned performance issues by comparing the anomaly with normal executions.

Detecting whether a given request is normal or abnormal is a challenging task. Some may assume that only a request response time is relevant to examine and, therefore, believe it is a trivial task. Yet, response time is not the sole factor to be considered. For example, two requests with the same time duration can have different internal behaviors (e.g., they generate different numbers of page faults, or they behave differently when they access the disk). Although they have identical response time, the one with an abnormal behavior can foreshadow an impending problem. Therefore, a general method of grouping requests based on their internal behavior is necessary. This grouping will be used for post-mortem root cause analysis.

This observation brings two research questions: (1) Can anomalies be efficiently and automatically detected from trace data? (2) Can the root cause of the anomaly be identified by comparing its internal run time behavior to that of a normal one? We restrict the scope of this paper to the requests, that we consider to be any task separated by a specific start and end event. Valid examples are web requests, database requests, and multiple calls to the same function in any application. In particular, we would like to discover which internal action or behavior causes a web request to run slowly.

Our main contributions are: (1) Tracing different levels of requests executions and extracting their internal behavior by following all notable threads along the critical path, rather than merely a single thread. (2) A proposed pipeline to detect outliers, and cluster them according to their run time behavior, which will enable comparison and evaluation. We show that the extracted features are especially useful for detecting outliers.

The remaining part of the paper is organized as follows. Section 5.3 presents the related work. Section 5.4 introduces our proposed approach to extract meaningful features, detect anomalies, and subsequently identify their root cause. Section 5.5 details our results and meticulously analyzes the complexity of the proposed approach. Finally, Section 5.6 concludes this work.

5.3 Related Work

Dynamic analysis through execution tracing is used to analyze software behavior [169, 170]. The analysis of kernel tracing data is studied in [171], while the processing of system call traces are studied in [169, 172]. The visibility of most of the existing works is, however, limited to only a single layer of the system, while one would typically need visibility of multiple layers (e.g., application, system calls, network, etc.) in order to completely analyze internal software behaviors [13]. In this paper, we analyze traces collected from both the application and kernel layers, further allowing for more effective insights into the software run time behavior.

As the complexity and variety of modern systems increase, so does the need to automatize their analysis. Recent progress in machine learning (ML) contributes interesting results for automating diagnosis tasks like intrusion and malware detection [2] or code correlation and optimization [173]. These techniques can be used to automatically detect changes in code [174], configuration, environment, run time behavior, and performance [175, 168]. Unfortunately, the variety of tracing formats, the large size of run time data, as well as their unstructured nature present additional challenges in both the data modeling and processing steps which require special care.

In the literature, kernel traces are represented in a variety of methods, but two primary approaches emerge. The first one preserves the ordering and the temporal information. These representations are sequences of system call names, learned system call embedding [2], and kernel states [13]. The second approach aggregates the sequences, trading the temporal information for a more compact representation. The main example of this approach is called bag-of-word, also known as *system call counts vector* [20], *frequency counts of system call names* [21] or *bag of system calls* [22]. N-gram is a technique which lies in between the two approaches as it preserves only a short and local ordering. This method has been extensively used for anomaly detection and intrusion detection as they are more expressive [21, 23, 25]. This work falls into the second approach, with the exception that duration is used in addition to counts.

Clustering anomalies based on their behavior helps to better understand, predict, and maintain them. Ideally, one would prefer to cluster anomalies based on their underlying and unknown root cause. A practical approach is to cluster them based on system resources consumption behaviors, thus grouping the faulty behaviors based on the extent to which they compete over resources to serve user requests. In a recent work, Nemati et al. [176] extracted high-level features from low-level traces of virtual machines and grouped them using two-stage k-means. Their method provided insights into the different behaviors and potential anomalies. Their work, however, relies on data collected from only one layer, which restricts the overall visibility.

This paper addresses the above-mentioned limitation by proposing an approach based on a multi-level data collection policy to ensure maximum visibility while undertaking the root cause analysis. Collecting data from different levels provides a more complete view of the system dynamics (application, operating system, network, etc.) and ensures that a broad range of anomaly can be detected. Indeed, although the application may be bug-free, a contention for resources between instances of the same application can be the source of inexplicable latencies. Such a complication is only visible from the operating system level.

Furthermore, our data collection process exceeds the simple tracing of single threads, which is the case in most of the aforementioned research works. Our method follows a request across all interacting threads, and possibly across different machines, to ensure that all required levels of details are collected.

Although using off-the-shelf, standard, unsupervised techniques, this new approach, with a pipeline to detect, cluster and analyze anomalies in requests, has not been proposed earlier, to the best of our knowledge.

5.4 Proposed Approach

5.4.1 Data Collection

Gathering applicable data is the first step in the proposed pipeline, as summarized in Figure 5.1. To that extent, we used the Linux Trace Toolkit Next Generation (LTTng) [177], a low-overhead open-source tracing tool that collects data from several layers. Data was acquired from the user space level, to distinguish the start and end of each request, as well as from the kernel level, to collect information regarding what actually occurs within the system during each request.

User space tracers typically work by instrumenting the source code (i.e., the web application source code) or the language core (i.e., the PHP core). In this work, the latter is instrumented because it adds a new tracing extension that inserts tracing macros in different entry points (function calls, request entry and exit, etc.). This ensures that users do not need to change their own source code. Each time the execution in the language core reaches a tracepoint, the associated macro is executed, generating an event that is stored in the corresponding CPU buffer. Table 5.1 shows all the events that the PHP tracer is able to collect. This PHP extension is open-source and publicly available[1].

Table 5.1 Tracepoints in the developed PHP tracing extension.

Trace Event	**Description**
request_start	Fires when a new PHP request is arrived.
request_exit	Fires when the handing of a PHP request is completed.
function_start	Fires when a function is called.
function_exit	Fires when a function exits.

Kernel tracers, on the other hand, usually work by instrumenting the different parts of the operating system. Luckily, the Linux kernel is already instrumented and has over two hundred incorporated tracepoints. The LTTng tracer attaches to these tracepoints and gathers

[1]https://github.com/naser/LTTng-enabled-PHP

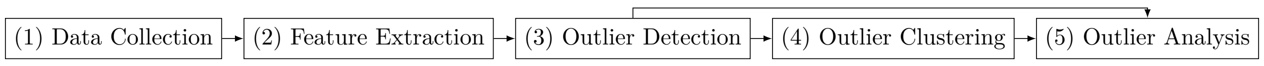


Figure 5.1 The proposed pipeline for anomaly detection.

data on system calls, processes, file systems, disk accesses, memory accesses, network layers, interrupts, timers, and other relevant areas.

5.4.2 Feature Extraction

The raw collected trace data is initially in a semi-structured, multi-dimensional, and multi-level format which needs to be projected in a lower-dimensional vector space before being analyzed. Indeed, most clustering algorithms accept only fixed-size arrays and are sensitive to the quality and compactness of the representation[2].

First, the trace must be divided into individual requests. The collected trace contains events from multiple web requests concurrently received by the server. Only two events from user space, namely `request_start` and `request_exit`, are required to correctly identify the boundaries of each request, received by a web server like Apache, and are passed on to the PHP Engine. Intuitively, `request_start` is fired when the PHP Engine accepts a request and `request_exit` when its handling is completed.

The next step is to collect detailed metrics from the kernel trace data to create a feature set for each request. A request can be handled by a single thread or, as it is the case with the majority of existing servers, by multiple threads. For instance, the latter can be the collaboration of a web server (e.g., Apache web Server or Nginx), a web application (e.g., PHP or Node), and a database (e.g., MySQL or SQL Server). Our analysis is based on features extracted from the critical path of each request, which spans across different threads. Figure 5.2 shows the critical path of a typical web request. The request time is broken into five intervals across four interacting threads, showing the detailed contribution of each thread to the total response time.

To discover the critical path of each request, we rely on the algorithm introduced by Giraldeau et al. [178]. They proposed an algorithm that would extract the active execution path across threads which possibly run on distant machines. Their method extracts various execution states for each thread, including running, interrupt handling, waiting for disk, waiting for network, waiting for timer, and waiting for another task. The algorithm first builds an execution graph showing all interactions between threads. Then the critical path – the

[2]See the curse of dimensionality for an example of a bad representation.

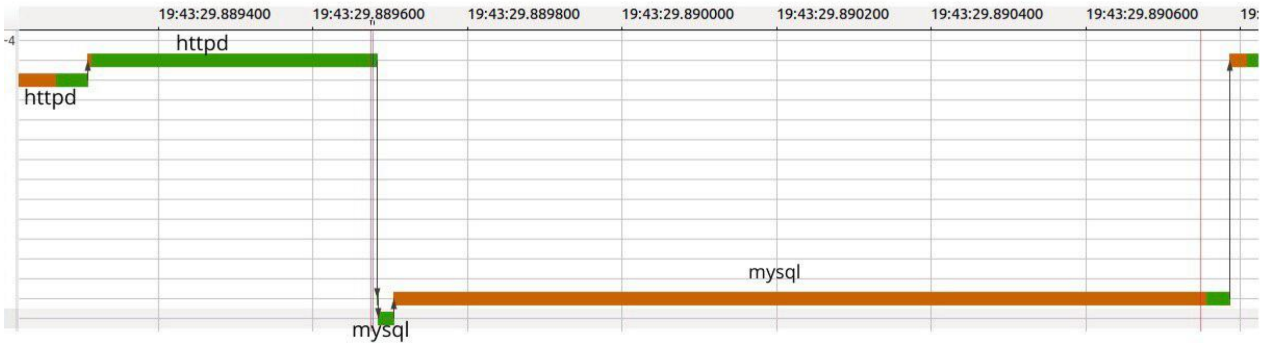


Figure 5.2 Critical path of a single web request spanning four threads.

minimal active path – is extracted by replacing recursively the waiting edges of a thread by the edges of the waking thread, from within the execution graph. In the final graph, for which an example is shown in Figure 5.2, vertical edges denote the waker/wakee relationships between the threads while the horizontal edges represent the different execution states within a thread.

The method proposed by Giraldeau et al. [178] was adapted to retrieve metrics regarding the different execution states that contribute to the response time of each request. Notably, the following execution states are considered:

- Running in system call mode (RS)
- Running in user mode (RU)
- Blocked for disk I/O (BD)
- Blocked for network (BN)
- Blocked for CPU (preempted) (BP)
- Blocked for another task (BT)
- Blocked for futex (BF)
- Blocked for interrupt (BI)
- Blocked for timer (BS)
- Total request time (TT)

Table 5.2 shows the four different feature sets that were created for each request based on the above execution states.

Table 5.2 Request feature sets used for clustering.

Feature set	Example
System calls sequence	`open, seek, read, close...`
Execution states sequence	`RU, RS, BD, BT, BP...`
Execution states count	15, 0, 1, 23, 110...
Execution states total duration	0.12, 0.0, 0.01, 0.85...

5.4.3 Outlier Detection

In order to automatically detect performance issues, one method includes first detecting outliers – out-of-distribution data points – then applying domain-specific tools to discover the root cause of the anomaly. However, the detection of outliers is highly dependent upon the data representation. In this paper, we propose a framework to cluster abnormal requests based on the features previously extracted. This section corresponds to the pipeline third step.

Clustering algorithms minimize the inter-point distance within a cluster while simultaneously maximizing the distance between clusters. However, choosing the *right* number of clusters is not a trivial task. Moreover, because the points of interest are abnormal, the method needs to provide an efficient clustering of outliers.

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [179] is a well-known clustering algorithm which does not require the number of clusters to be specified. Instead, it accepts two parameters: ϵ the maximum distance for two points to be considered neighbors and m the minimum number of points required in order to create a cluster. DBSCAN works by first selecting a point X at random. If X has at least m neighbors, then a cluster is created with X and all of its neighbors. The cluster is expanded by then considering the neighbors of X. If X has less than m neighbors and is not part of a cluster, then it is considered an outlier. The process continues until all points are considered.

DBSCAN presents three key advantages: (1) the number of clusters is implicit, (2) an arbitrary shape of clusters are created, and (3) outliers are automatically separated. However, DBSCAN is sensible to its two parameters – ϵ and m. Those parameters can be set manually or estimated with a random search. Furthermore, DBSCAN can only be applied to multi-dimensional data of a fixed size. Thus, the variable-length sequences of system calls and features must be converted to a fixed size array.

One approach to representing a variable-length sequence with a fixed size vector is to count the number of occurrences of each event. This technique is called "bag-of-word" (BoW) and has been extensively used in natural language processing. One may argue that events that appear across all executions carry little information. The term "frequency-inverse document frequency" (TF-IDF) is a popular technique to measure bag-of-word according to the importance of each event. More formally:

$$\text{tf-idf}_{i,j} = \text{tf}_{i,j} \times \log\left(\frac{|D|}{|d_j : t_i \in d_j|}\right) \tag{5.1}$$

where $\text{tf}_{i,j}$ is the frequency of the i-th event in the j-th sequence, $|D|$ is the total number of sequence, and $|d_j : t_i \in d_j|$ is the number of sequence containing the i-th event.

Specific methods such as Markov Models have been developed to detect outliers within sequences. However, those methods assume that events are regularly generated, (i.e., there is a fixed duration between events). This crucial assumption does not hold true in the context of traces. Moreover, those techniques are computationally expensive. For those reasons, they are not included in the proposed framework.

5.4.4 Outlier Clustering and Analysis

Outliers are rare by definition, yet they may be numerous enough to require clustering before they can be analyzed. This corresponds to the optional fourth step in our pipeline.

The objective is to separate different types of abnormal behavior. To that extent, feature counts are more informative than feature duration. Indeed, a slow request has either a high count of non-blocking events or a small count of blocking ones. However, consider that a fast request may also have a high number of non-blocking system calls like `fnctl`[3], surely indicating that something is behaving incorrectly.

K-means is a simple yet efficient clustering algorithm that has been widely used because of its ability to find tight spherical clusters. However, k-means is a distance-based method which means it is sensitive to data scale. To mitigate this effect, the feature counts have been standardized. The elbow method [180] was applied in order to find the appropriate number of clusters. This is an ambiguous heuristic stating that one should choose the number of clusters, as long as it does not substantially decrease the inertia.

Once outliers have been clustered, each group is separately analyzed. This is the last step in our pipeline. As an example of a typical analysis, statistics about requests and average n-grams were computed independently, for each cluster of outliers as well as for non-outlier

[3]`fnctl` is the system calls that manipulate a file descriptor. For more information, see: `http://man7.org/linux/man-pages/man2/fcntl.2.html`

requests. Finally, we were able to link those results with real anomalies, confirming that the proposed pipeline is indeed able to detect outliers, group them appropriately, and bring forward useful information regarding the root cause of the anomaly.

5.5 Results

The complete project including the code, data, parameters, and results is available on GitHub[4].

As a proof of concept, around 50,000 requests were generated using the Apache benchmark suite[5]. The experiment was performed using anywhere from 1 to 1,000 clients. In the PHP tracing module, only the two tracepoints that collect the start and end of each request were enabled. This gave a window for which we collected detailed kernel events.

5.5.1 Outlier Detection

Once requests were generated and features extracted, DBSCAN was applied to the different representations discussed in section 5.4. The parameters ϵ and m were set manually for each experiment.

The outlier detection was first evaluated qualitatively by comparing the distribution of durations for outlying and non-outlying requests. One would expect little overlap between the two groups as we know that slow requests are abnormal, hence outliers, and that fast ones are largely normal.

System call counts and feature counts did not provide a clear separation of the two distributions (Fig. 5.3). TF-IDF did not yield better results, because some events are so common that their weight is almost null. Only the feature duration was able to provide a clear separation, with little overlap between the outliers duration distribution and the non-outliers.

The outlier detection was then evaluated quantitatively by looking at two statistics:

1. The median duration of outlying requests. The median is preferred to the mean as it is less sensitive to a few miss-classified normal requests. One would expect the median duration of outliers to be significantly higher than for the whole data set.

2. The probability of a detected outlying request having a duration above 200, 250, and 300 milliseconds. Those values are possible thresholds to detect slow requests, also

[4]`https://github.com/qfournier/request_analysis`
[5]`https://httpd.apache.org/docs/2.4/fr/programs/ab.html`

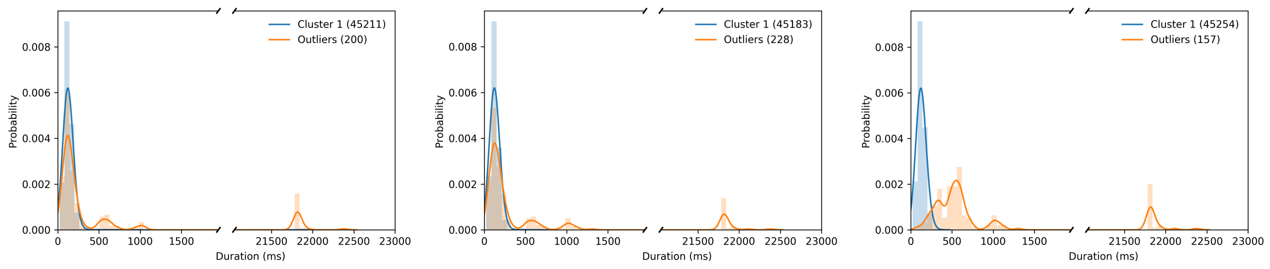


Figure 5.3 Distribution of the duration of outlying and non-outlying requests. DBSCAN was applied on three different requests representations: (left) system call counts, (middle) feature counts, and (right) feature duration. The size of each cluster is specified between parenthesis in the legend.

known as anomalies. One would expect the probability to be high, but not equal to 100% as some fast requests could also have an abnormal behavior.

Table 5.3 Outlying requests' statistics.

	System calls		Features		
	BoW	TF-IDF	BoW	TF-IDF	Duration
Number of outliers	200	362	228	236	157
Median duration	147	115	168	155	**554**
$P(d > 200)$	0.36	0.163	0.408	0.390	**0.968**
$P(d > 250)$	0.28	0.108	0.316	0.305	**0.924**
$P(d > 300)$	0.26	0.940	0.276	0.267	**0.892**

DBSCAN yielded an outlier detection rate lower than one percent for each representation, which is reasonable (Table 5.3). The median duration for the whole data set is 122 ms, which is close to every outliers median duration, except for outliers detected from the feature duration. Those same outliers have a probability of being slower than 250 ms of 92.4%, which is about what is expected. The quantitative results concur with the qualitative results: DBSCAN works more efficiently with feature duration to detect outlying requests. Only those outliers are considered in the rest of this paper. For simplicity, the non-outlying requests – the ones not detected as outliers by DBSCAN – are called *normal* requests even if they may contain anomalies.

As a short ablation study, DBSCAN was replaced with another outlier detection method called isolation forest. Similar results were obtained, and the same conclusions could be drawn. This indicates that it is not merely a preference of DBSCAN but rather the feature duration that is retaining useful information to detect outlying requests.

5.5.2 Outlier Clustering and Analysis

K-means was applied to the feature counts of the outliers detected with DBSCAN. Two to ten clusters were studied. Using the elbow rule, three clusters were selected.

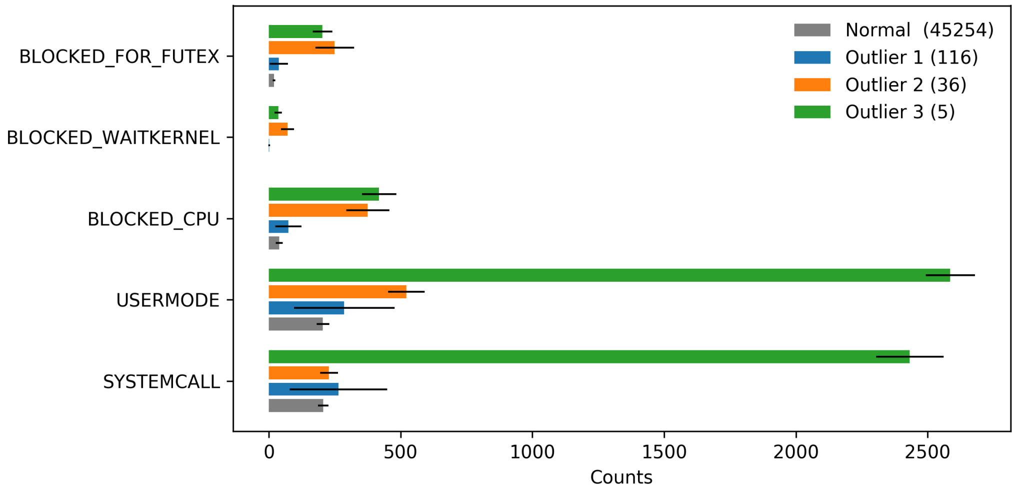


Figure 5.4 The average feature count in every cluster. The standard deviation is represented with a horizontal black line. There are three different types of outliers which depend on different resources.

Table 5.4 Statistics of each cluster.

	Normal	Cluster 1	Cluster 2	Cluster 3
Duration (ms)	126	479	**15512**	559
Number of system calls	246	312	298	**2690**
Distinct system calls	26.0	26.4	26.1	**29.0**

The clustering can be visualized by plotting the feature count histogram of each cluster and of normal requests (Fig. 5.4). In addition, Table 5.4 compares statistics relating to requests in each group. Even though requests in the first cluster (blue) are nearly four times slower than the normal clusters, feature counts alone are not sufficient to set them apart. The high standard deviation of `usermode` counts indicates that this cluster is heterogeneous. The second cluster (orange) is composed of the slowest requests, which have a higher count than the normal ones, especially for `blocked` states. Finally, the last cluster (green) is characterized by an extremely high number of counts, especially for `usermode` and `systemcall` states. Although these are not the slowest requests, they have on average ten times more system calls and three more distinct system calls than the normal requests.

Table 5.5 shows differences in average n-grams between the normal requests and each cluster. They have been handpicked as they are unique to each cluster and explanatory of the

Table 5.5 Average unigrams, bigrams, and trigrams distinctive of each cluster.

Average n-gram	Normal	Cluster 1	Cluster 2	Cluster 3
`write`	1.0	**2.3**	1.1	1.6
`write write`	0.0	**1.3**	0.1	0.6
`write write write`	0.0	**0.9**	0.1	0.4
`cnnct`	1.0	1.0	**5.3**	1.0
`cnnct cnnct`	0.0	0.0	**4.3**	0.0
`cnnct cnnct cnnct`	0.0	0.0	**3.7**	0.0
`fcntl`	7.0	7.0	7.0	**419.0**
`fcntl fcntl`	2.0	2.0	2.0	**305.0**
`fcntl fcntl fcntl`	0.0	0.0	0.0	**193.0**

underlying problem. Note that `connect` is shortened as `cnnct`. The complete list of n-grams is available on GitHub.

Requests in the first cluster differ from the normal ones only by their number of `write` system calls. It is not sufficient to draw any conclusion. One would need to apply domain-specific methods in order to determine the root cause or to show that those requests are false positives – outliers that have normal behavior.

Requests in the second cluster are characterized by a high number of `connect` and its repetition. Note that if the flag `O_NONBLOCK` is not set, `connect` is a blocking system call. This indicates that PHP cannot initiate a connection on a socket.

The third cluster is compelling as it explains cache issues. Further investigation into the PHP engine source code reveals that the `fcntl` system call is used for locking purposes. This, in fact, shows that there are unexpected blocking and waiting occurrences, hence some issues in the PHP processes while handling specific requests. At its core, PHP employs a cache mechanism called OPcache (OpCode Cache) to cache compiled script byte-codes in a shared memory for improved request handling performance. Whenever a script is compiled, the process checks the OPcache on the shared memory for an already-compiled code. If the code is missing, the PHP process compiles the code and writes the resulting byte-code to memory. However, when one process is writing into the shared memory, other processes must wait for the first one to release the lock before obtaining access and writing in the OPcache itself. This was the case for requests in the third cluster: PHP processes blocked each other to access the OPchache shared memory. A detailed analysis using a Critical Path View [171] in Trace Compass (an open-source trace analysis tool[6], confirms that there are issues among the PHP processes (shown in Figure 5.5) when concurrent requests are handled in the server.

[6]https://tracecompass.org

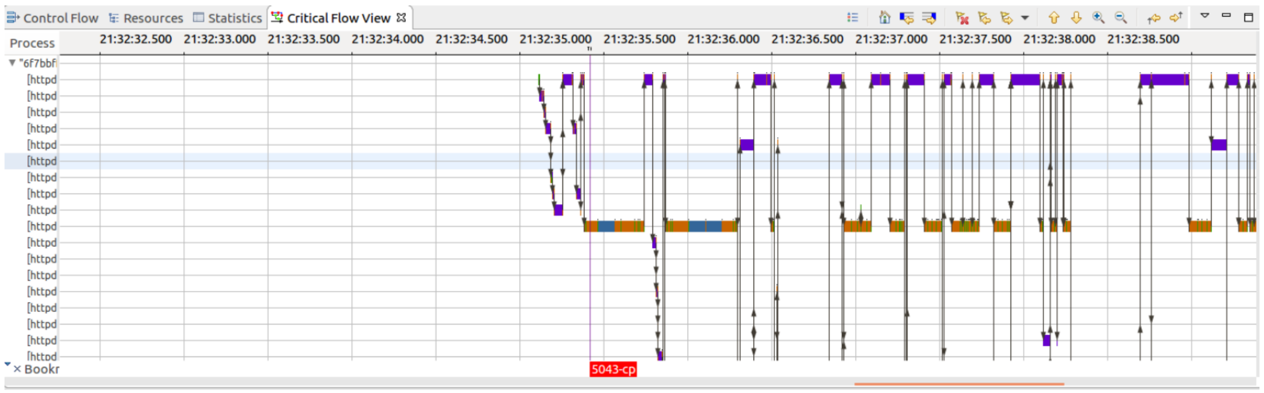


Figure 5.5 Critical Path analysis showing the contention between PHP processes.

5.5.3 Visualization

A qualitative display of the feature-duration space was obtained by reducing the dimensionality using isometric feature mapping (Isomap) [181]. Isomap is a non-linear dimensionality reduction method that learns an embedding that preserves the intrinsic geometry of the data. This projection can be utilized to set a threshold duration, after which all requests are considered abnormal, speeding up outlier detection. As the cluster gathers requests up to 300 ms, it is a reasonable choice.

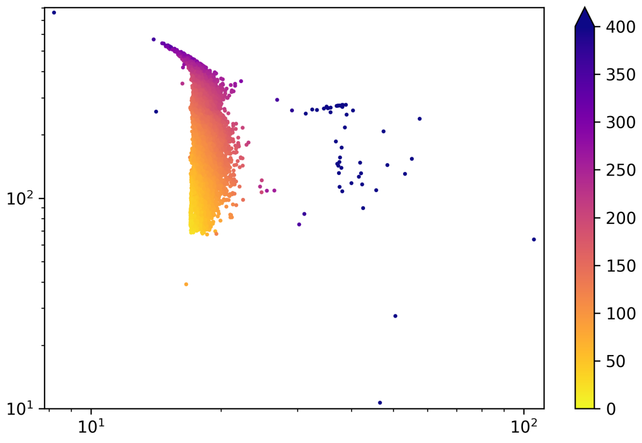


Figure 5.6 Visualization of the feature-duration space using ISOMAP. The color indicates the duration of the request in milliseconds. There are about 10 clear outliers outside of the window. Note the log scale on both axes.

5.5.4 Computational Cost

The proposed method for clustering normal and abnormal requests, based on kernel-level features from thread interaction in the critical path, will prove useful only if it can be efficiently scaled.

Experiments were conducted on a modern computer equipped with a 6-core processor and 32 GB of main memory. Requests were generated using a Wordpress website running on Linux 18.04 with PHP 7.0.32 and Apache 2.4.18. The trace was collected using LTTng 2.11.

Tracing Cost

Experiments were performed using between 1 and 1,000 clients with the following tracing configurations:

- No tracing.
- Minimal tracing: only a small subset of events required for analysis is enabled.
- Full tracing: all kernel space and user space events are enabled.

The capacity of the server to handle requests per second is shown in Figure 5.7, using the above tracing configurations. When all kernel space and user space events are enabled, tracing has a significant impact on the performance, with a slowdown of $29.6 \pm 2.0\%$. This is mostly due to the kernel tracing, since it collects and stores a large amount of system execution details to the disk. However, the proposed analysis does not require enabling all the tracepoints. In user space (i.e., PHP core), only two events are enabled: *request_start* and *request_end*. In kernel space, only scheduling, system call, timer, and waking up events are necessary to extract all the essential execution states. With this minimal setup, the overhead is reduced to $5.1 \pm 1.9\%$, coming mostly from the kernel-enabled events.

Analysis Cost

Note that the analysis is written in Python and is not parallelized unless explicitly specified. Much faster run times could be achieved with a C or C++ implementation.

Creating the feature bag-of-word took 1.17 ± 0.12s, as it is linear in the number of examples. It is not included in Table 5.6 since the feature duration was used instead. Furthermore, as the bag-of-word matrix is often sparse, it can be efficiently stored in memory.

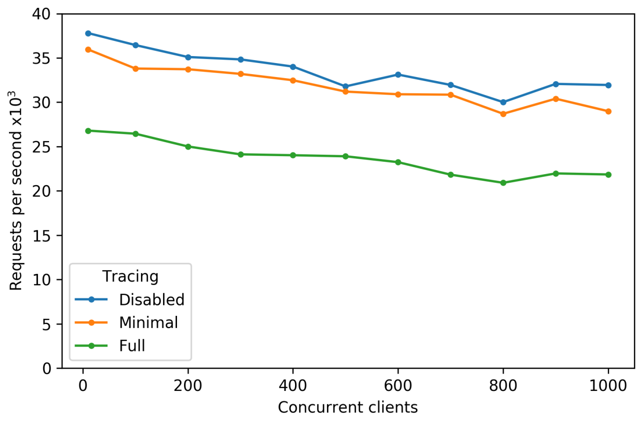


Figure 5.7 Cost of the different tracing configurations, measured by the number of request per second.

Table 5.6 Average run time of each step plus or minus the standard deviation. Each step was run ten times.

Step	Run time
Feature extraction	29.31 ± 1.08s
Outlier detection	5.69 ± 0.83s
Outlier clustering	0.04 ± 0.00s
N-gram analysis	17.59 ± 0.16s

The run time complexity of DBSCAN is $\mathcal{O}(n\text{log}n)$, assuming it uses an indexing structure and an ϵ which gives log$n$ neighbors an average [179]. The parallelized scikit-learn implementation of DBSCAN was used.

K-means is known to be an efficient method, with an average complexity of $\mathcal{O}(knT)$ where k is the number of neighbors and T is the number of iterations. The parallelized scikit-learn implementation of k-means was also used.

Even though n-gram is merely counting, and there are a limited number of possibilities (31 unigrams, 135 bigrams, and 334 trigrams in detected outliers), this is the slowest step as the results were stored in a data frame before being displayed and saved. This makes the data manipulation easier, although it is not necessary and can be optimized for production.

5.6 Conclusion and Future Work

In this paper, we introduced a pipeline for the automatic detection of performance issues in web requests. This includes both user space and kernel space feature extraction methods, which proved to be useful for detecting outliers and clustering them in a relevant manner.

To answer our first research question: it is indeed possible to automatically and efficiently detect anomalies from trace data. A simple statistical analysis was able to guide us to the root cause of two different anomalies. This indicates that comparing run time behavior gives meaningful insights into the root cause.

As future work, we would like to compare our method with ones specifically designed for sequences, even if some assumptions are broken. Notably, Hidden Markov Models and SEQDBSCAN – an extension of DBSCAN for sequences – seem promising. We would also like to extend the cluster analysis to context calling trees (CCT) or enhanced context calling trees (ECCT) [168] in order to determine if the first cluster in Figure 5.4 and Table 5.4 is a group of normal requests with a different behavior, or merely a group of unknown anomalies.

5.7 Acknowledgment

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson, Ciena, and EffciOS for funding this project.

CHAPTER 6 ARTICLE 3: ON IMPROVING DEEP LEARNING TRACE ANALYSIS WITH SYSTEM CALL ARGUMENTS

Authors Quentin Fournier, Daniel Aloise, Seyed Vahid Azhari, and François Tetreault.

Published at IEEE/ACM 18th International Conference on Mining Software Repositories 2021 (19 May 2021)

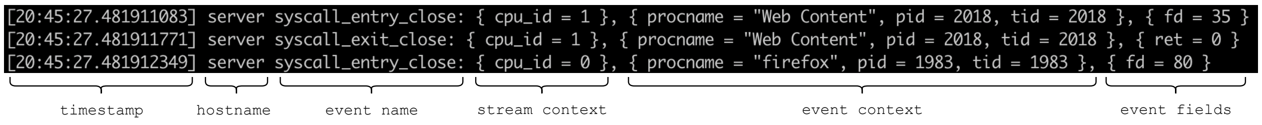
Abstract Kernel traces are sequences of low-level events comprising a name and multiple arguments, including a timestamp, a process id, and a return value, depending on the event. Their analysis helps uncover intrusions, identify bugs, and find latency causes. However, their effectiveness is hindered by omitting the event *arguments*. To remedy this limitation, we introduce a general approach to learning a representation of the event names along with their arguments using both embedding and encoding. The proposed method is readily applicable to most neural networks and is task-agnostic. The benefit is quantified by conducting an ablation study on three groups of arguments: call-related, process-related, and time-related. Experiments were conducted on a novel web request dataset and validated on a second dataset collected on pre-production servers by Ciena, our partnering company. By leveraging additional information, we were able to increase the performance of two widely-used neural networks, an LSTM and a Transformer, by up to 11.3% on two unsupervised language modelling tasks. Such tasks may be used to detect anomalies, pre-train neural networks to improve their performance, and extract a contextual representation of the events.

Keywords Tracing, Machine Learning, Deep Learning.

6.1 Introduction

In recent years, deep learning has been successfully applied to an ever-growing range of supervised and unsupervised tasks. This trend has been enabled by the ever-increasing computational resources and the novel techniques introduced to take advantage of these resources. As of today, the largest model for natural language processing (NLP) comprises 175 billion parameters and has been trained on half a terabyte of curated text [50]. The authors showed that the model performance scales consistently with the number of parameters and the amount of available data.

A technique that surely generates a large amount of data is *tracing*. Tracing is the act of collecting a trace which is a sequence of low-level events. Such events are produced whenever a specific instruction called tracepoint is encountered at runtime and comprises a name, a precise timestamp, and possibly many arguments. Figure 6.1 depicts three trace events.



```
[20:45:27.481911083] server syscall_entry_close: { cpu_id = 1 }, { procname = "Web Content", pid = 2018, tid = 2018 }, { fd = 35 }
[20:45:27.481911771] server syscall_exit_close: { cpu_id = 1 }, { procname = "Web Content", pid = 2018, tid = 2018 }, { ret = 0 }
[20:45:27.481912349] server syscall_entry_close: { cpu_id = 0 }, { procname = "firefox", pid = 1983, tid = 1983 }, { fd = 80 }
```

Figure 6.1 Trace events as displayed by Babeltrace[2]. The arguments are all the values except the event name. In this example, the arguments are from left to right: the timestamp, the hostname, the CPU id, the process name, the process id, the thread id, the file descriptor, and the return value.

Traces provide insights on the execution of a piece of code and have been extensively used to detect intrusions, identify bugs, and find the root cause of latency issues. The main advantages of tracers are (1) they do not require to stop the execution contrary to debuggers, and (2) they do not aggregate events or metrics contrary to loggers.

In this paper, we consider the events generated by the operating system also known as kernel events. The benefit of such events is two-fold: (1) tracepoints are already implemented in the Linux kernel, which allows tracing virtually any Linux system without having to modify the source code, and (2) the behaviour of the whole system is visible from the kernel. In this paper, we focus on a subset of the kernel events called *system calls*. System calls are the only way for an application to communicate with the operating system.

Although the manual inspection of traces may reveal insights that are virtually impossible to extract automatically, the amount of human labour required is often prohibitive. Indeed, the operating system produces thousands of events every second, most of which may be collected. The sheer size of traces is the primary reason why automatic analysis is required. Traces are used to detect *unknown* intrusions, to identify *unknown* bugs, or to locate the *unknown* root cause of anomalies, making their analysis often challenging to specify in practice. Therefore, machine learning techniques, that is, techniques that learn how to solve a task from examples, are well suited to analyse traces.

Most machine learning methods take a vector of numerical features as input. Hand-crafted features of traces have been proposed, but no representation seems to work universally well or to encapsulate the true underlying explanatory factors [13, 176, 27]. Instead of relying on hand-crafted features, neural networks learn how to extract meaningful features for the task.

[2]`https://babeltrace.org`

By finding a relevant input representation for the task, neural networks reduce the need for an expert, and the model performance is improved in most cases.

Although a wide range of deep learning techniques has been applied on traces by previous works, only a small fraction of the accessible information has been considered. The event arguments and, in certain cases, the event ordering inside the trace, have been left out in the literature. Section 6.2 discusses in more detail the related works and their limitations. We argue that the increase in resources and the improvement of deep learning techniques allow fully exploiting traces.

A trace is a sequence of discrete values arguably comprising a syntax and a semantic. Due to their resemblance to natural language, the most common approach is to apply deep learning techniques from natural language processing. Our methodology follows the previous works by considering the widely used Long Short-Term Memory (LSTM) [38]. A recent alternative to LSTM for processing variable-length sequences called the Transformer [43] is also evaluated. Although this model is omnipresent in NLP, it has not yet been applied on traces. The two models were evaluated on two unsupervised objectives: (1) left-2-right language model (LM) that allows computing the likelihood of a sequence, and therefore, detecting anomalies, and (2) masked language model (MLM) that is used for pre-training [51].

This paper's first contribution is the introduction in Section 6.3 of a novel method to learn a single representation of the system call names with their arguments. Results are detailed in Section 6.5 and an ablation study is conducted to investigate the impact of three groups of arguments: call-related, process-related, and time-related.

The second contribution of this paper is the introduction of a novel dataset comprising around 250,000 web requests. The actual dataset is provided, but most importantly, the data generation methodology is explained in Section 6.4. One may argue that our dataset is too simple or that it inaccurately represents actual web servers. Therefore, every experiment is validated on a second dataset collected on pre-production servers by Ciena, the partnering company of this research.

Finally, Section 6.6 discusses the possible threats to validity, and Section 6.7 answers interesting questions about the pertinence of the proposed approach and future works.

6.2 Related Work

Over the last two decades, a wide range of machine learning techniques has been applied to analyze traces, including naive Bayes [34], random forest [36], and hidden Markov models [24]. Recently, the trend has shifted toward more flexible approaches, and especially toward deep learning methods. Model flexibility relates to the space of functions that the model is able to learn and increases with the number of parameters. Therefore, highly flexible methods, such as large neural networks, are able to learn complex solutions that typically perform better than less flexible ones. This section provides an overview of the main neural networks that have been studied in the tracing literature as well as their limitations.

Recurrent neural networks (RNNs) allow processing variable-length sequences with a fixed number of parameters. Such a network produces an output at every time step and is depicted in figure 6.2. The Long Short-Term Memory (LSTM) [38] is a recurrent neural network specifically designed to learn dependencies across a large number of time steps. This network has been extensively and successfully used across many fields. Tracing is no exception, and LSTM is by far the most popular neural network to analyze traces [36, 20, 2, 17, 19].

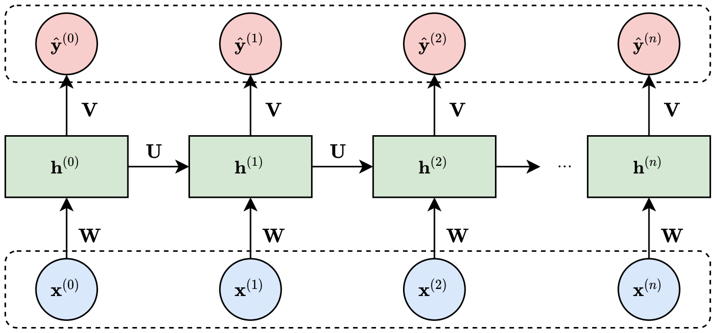


Figure 6.2 The unrolled computational graph of a recurrent neural network. The input and output sequences are depicted in blue and red, respectively. The time step is indicated in exponent and between parenthesis. Note that the network parameters W, U, and V, are replicated at every time step. Therefore, the network can process variable-length input sequences.

Dymshits et al. [20] trained a unidirectional and a bidirectional LSTM on *sequences of system call count vectors*. Such vectors are bag-of-words, that is to say, the normalized counts of system call names, from a fixed-duration window. This aggregation is a trade-off between computational efficiency and performance, and is controlled by the window size. The authors also trained an Inception-like net consisting of multiple LSTMs with tied weights. They found that simpler LSTMs performs on par with the more complex ones.

Kim et al. [2] trained an ensemble of LSTMs on sequences of system call names. Ensemble techniques improve the performance, although not significantly, and the robustness of the chosen method. While ensemble techniques may be necessary for industry products, this paper will not leverage them as the main objective is to show the relative impact of the arguments rather than the approach's absolute performance.

Song et al. [36] compared an LSTM with less flexible machine learning techniques to detect and explain anomalies from streams of traces. They did not, however, explicitly say which events were considered or describe their preprocessing.

Recurrent neural networks output a vector at every time step, so the output sequence must have the same length as the input sequence (see Figure 6.2). This property of RNNs may become a constraint depending on the task. To overcome this limitation, Sutskever et al. [40] introduced the sequence-to-sequence framework where a first network (encoder) encodes the input sequence into a fixed-size context. A second network (decoder) then generates the output sequences based on this context. This framework allows outputting a variable-length sequence independently of the input sequence length and is illustrated in Figure 6.3.

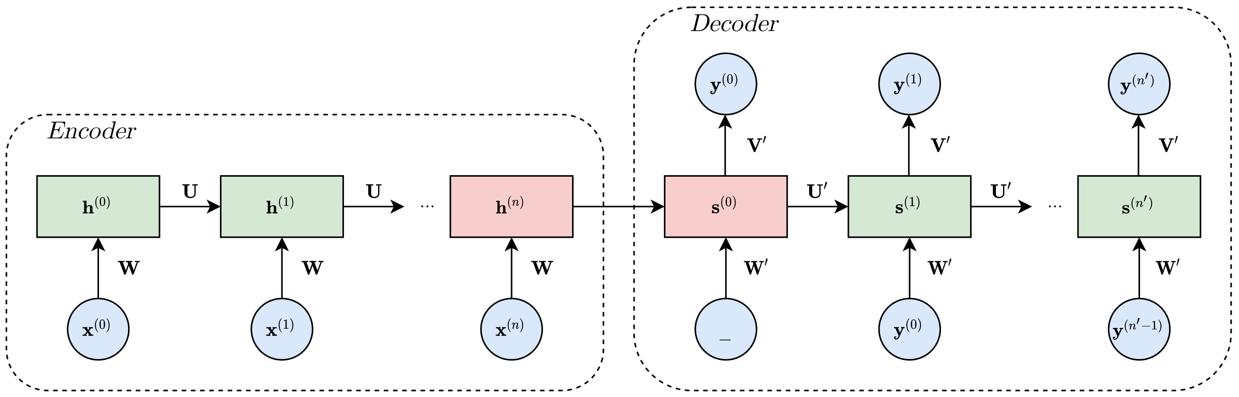


Figure 6.3 Sequence-to-sequence framework. A first network (encoder) encodes the input sequence into a fixed-size context $\mathbf{h}^{(n)}$ shown in red, then a second network (decoder) generates the output sequences based on this context.

Lv et al. [26] used a gated recurrent unit[3] (GRU) [39] in a sequence-to-sequence fashion to extend sequences of system calls names and increase the accuracy of intrusion detection.

Recurrent networks, including LSTMs and GRUs, suffer from an issue related to memory compression [41]. As the input sequence gets processed, information must be stored in the fixed-size hidden representation $\boldsymbol{h}$. Either $\boldsymbol{h}$ is too large and computational resources are wasted, or $\boldsymbol{h}$ is too small and information is lost. In the latter case, the model performance might be significantly impacted. Bahdanau et al. [42] introduced an alignment mechanism called *inter-attention* to mitigate the effect of memory compression. This mechanism computes a different representation of the input for each output step, effectively allowing the decoder to "look at" the relevant part(s) of the input for each output step. Figure 6.4 illustrates the inter-attention mechanism.

[3]GRU is similar to LSTM but requires fewer parameters.

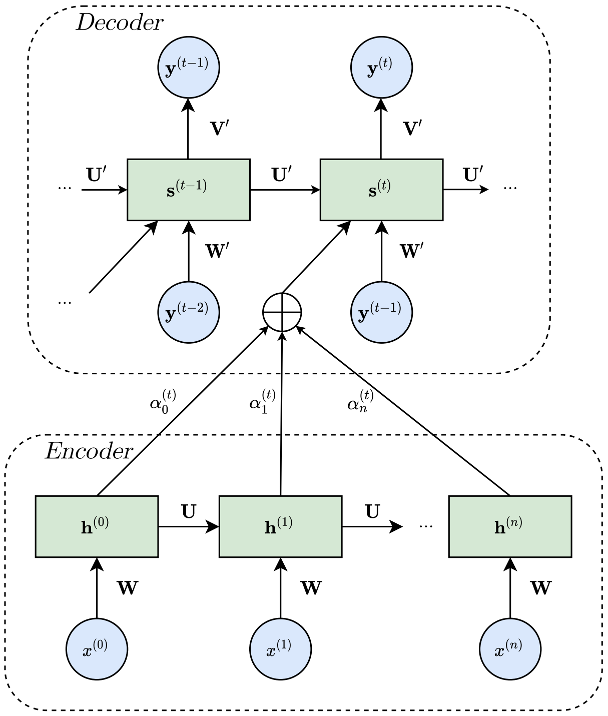


Figure 6.4 Inter-attention mechanism. The attention weight $\alpha_i^{(t)}$ corresponds to the strength with which the i-th encoder hidden representation $h^{(i)}$ contributes to the context of the t-th decoder step.

Brown et al. [4] augmented an LSTM with the dot-product inter-attention and explored different ways of computing the attention: fixed, position-based (syntax attention), and context-based (semantic attention). For the task of system log anomaly detection, every attention yielded comparable results.

Finally, due to their sequential nature, recurrent networks do not scale efficiently to longer sequences [43]. Dai et al. [48] introduced the relative effective context length (RECL), the largest context length that leads to a substantial relative gain over the best model. Simply put, increasing the context length over the RECL yields a negligible increase in performance; thus, RECL indicates the maximum dependency length that the model is able to learn. They showed that the RECL of LSTM is limited to around 400 time-steps. This is problematic for trace analysis since hundreds of events may be generated every second.

To overcome this limitation, Vaswani et al. [43] introduced the Transformer, a sequence-to-sequence model based solely on the inter-attention and self-attention mechanisms. The self-attention allows relating any two positions in a sequence regardless of their distance thus allowing for a significant increase in performance in most natural language processing tasks at the cost of a quadratic complexity with respect to the sequence length. To the best of our knowledge, this model has not been applied on traces but has been included considering its ubiquity in NLP.

None of the aforementioned works considered the system call arguments. Arguably, the main reason is that the community does "*[...] not have a compact fixed-dimensional representation for system call arguments suitable for large-volume training and classification.*" Dymshits et al. [20].

Nonetheless, Nedelkoski et al. [17] used a bimodal LSTM that is the concatenation of two LSTM hidden representations trained on the real-valued duration and one-hot-encoded texts, respectively. Albeit their work considered logs rather than traces, one may view their method as leveraging a temporal argument. The neural network proposed by Ezeme et al. [19] is the closest to actually considering multiple arguments values. The authors trained an LSTM using the system call name, the CPU cycles count, and the distribution of characters in the arguments' values.

As far as we know, only two works by Tandon and Chan [15, 15] considered the actual values of multiple system call arguments. The authors trained a conditional rule-learning algorithm called LERAD, which, contrary to neural networks, does not require to learn a fixed-size representation of the arguments.

6.3 Proposed Approach

Before introducing the proposed approach, let us clarify the different categories of system call arguments. One may group them depending on whether they are part of the stream context, the event context, or the event fields (see Figure 6.1). In this work, the arguments are grouped based on their semantic. The first category comprises all *call-related* arguments such as the return value, the file descriptor, the type of futex operation, and the number of bytes to write – depending on the event. The second category consists of all *process-related* arguments such as the process name, the thread id, and the process id. Note that this category corresponds exactly to the event context. Finally, the third group consists of *time-related* arguments such as the timestamp and the timeout duration.

The scope of this work is limited to the arguments that are common to virtually all system calls. Namely, the return value (`ret`), whether the event corresponds to the start or end of a system call execution (`entry`), the process name (`procname`), the thread id (`tid`), the process id (`pid`), and the timestamp (`timestamp`). As explained later, extending this work to other arguments is simple but may require a substantially larger dataset. Table 6.1 recapitulates the considered arguments.

Table 6.1 The studied system call arguments.

Category	Argument	Notation	Type
call-related	return value	`ret`	integer
	start/end of execution	`entry`	boolean
process-related	process name	`procname`	string
	process id	`pid`	integer
	thread id	`tid`	integer
time-related	timestamp	`timestamp`	integer

In order to determine how to represent the arguments, one must identify the intrinsically meaningful ones. In other words, one has to assess whether the argument values convey meaning in themselves – without any context. As an example, let us consider the process name "apache". This value means that an Apache web server has generated the system call, hence `procname` is inherently meaningful. On the contrary, the process id "12523" is only meaningful in the context of the trace. Indeed, the `pid` allows relating events that have been generated by the same process; the value "12523", however, may well be associated with two distinct processes at different points in time.

The `procname`, the `return value`, and the `entry` are intrinsically meaningful arguments, and hence, an embedding will be learned for them. On the contrary, the `pid`, the `tid`, and the `timestamp` are not inherently meaningful and an encoding will be applied.

6.3.1 Embedding

One way to represent textual words is through a sparse binary vector called one-hot-encoding. The i-th word of the vocabulary is mapped to a row vector $\boldsymbol{e}_{w_i}$ whose dimension is equal to the size of the vocabulary. Such vector is filled with 0 except for the i-th position which is equal to 1. Given a toy vocabulary of three system call names $\{\texttt{open}, \texttt{close}, \texttt{timer}\}$, their one-hot-encoding would be $[1, 0, 0]$, $[0, 1, 0]$, and $[0, 0, 1]$, respectively.

One-hot-encoding has two major drawbacks: (1) the vector dimension is equal to the vocabulary size which may be large, and (2) the encoding of any two distinct words are perpendicular, meaning that words are equidistant. For instance, one would expect $\text{dist}(\boldsymbol{e}_{\texttt{open}}, \boldsymbol{e}_{\texttt{close}}) < \text{dist}(\boldsymbol{e}_{\texttt{open}}, \boldsymbol{e}_{\texttt{timer}})$ as `open` is semantically closer to `close` than to `timer`.

A better representation is expected to be more compact and to encapsulate semantic knowledge about the word. Such representation is called an *embedding*. Note that in the natural language processing community, an embedding refers to both the general mapping from a textual space to a semantic vector space and the actual dense vectorial representation of a word.

Formally, an embedding is defined by a dense matrix $\boldsymbol{W} \in \mathbb{R}^{d_v \times d_e}$ with d_v the size of the vocabulary and d_e the dimension of the embedding such as $d_e \ll d_v$. The embedding $\boldsymbol{x}_{w_i}$ of the word w_i is computed by multiplying its one-hot-encoding $\boldsymbol{e}_{w_i}$ with the embedding matrix $\boldsymbol{W}$ which effectively acts as a lookup table (see example below). The embedding matrix is typically treated as any other model parameter in that it is randomly initialized and learned with gradient descent.

$$\underbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}}_{\text{One-hot vector } \boldsymbol{e}_{w_i}} \times \underbrace{\begin{bmatrix} 5 & 6 & 2 & 1 & 4 \\ 0 & 1 & 7 & 3 & 1 \\ 4 & 8 & 1 & 6 & 9 \\ 3 & 1 & 2 & 8 & 2 \end{bmatrix}}_{\text{Embedding matrix } \boldsymbol{W}} = \underbrace{\begin{bmatrix} 4 & 8 & 1 & 6 & 9 \end{bmatrix}}_{\text{Word embedding } \boldsymbol{x}_{w_i}}$$

6.3.2 Encoding

It would be ill-advised to learn an embedding of a value that is not inherently meaningful – whose interpretation depends entirely on the context. Instead, one should use a deterministic transformation without any parameter that is called an encoding.

Once more, let us consider the process id. Neural networks take as input a vector of numerical values. Therefore one may provide the actual `pid` as input. It is, however, a best practice to normalize the input vector to mitigate numerical instabilities, help training, and improve the model performance. Since the `pid` is not inherently meaningful in general[4], any bijection from the argument space to a small interval such as $[0, 1]$ or $[-1, 1]$ works well. The simplest solution would be to map the `pid` uniformly to real values between $[0, 1]$. In practice, the number of distinct `pid` within a trace varies and is often unknown beforehand.

A practical way to encode a numerical value is to apply the cosine function. Indeed, the codomain is $[-1, 1]$, and the function requires no knowledge about the distribution or the extremum of the input variable. The cosine function is not, however, a bijection. As a result, collisions may occur: two different values assigned to the same encoding. Consider $x = 1$ and $x' = 1 + 4\pi$:

$$cos(1) = cos(1 + 4\pi)$$

The number of collisions may be reduced by dividing x by an appropriately large number

[4]There are exceptions. Notably, `pid` 0, `pid` 1, and kernel-reserved `pid`s are meaningful and could be considered separately.

which effectively controls the period of the cosine function. Note that if the denominator is too small, collisions may still occur.

$$cos(1/2) = cos((1 + 4\pi)/2)$$

If the denominator is too large, the encodings will be extremely close, hence difficult for a model to distinguish.

$$cos(1/1000) \approx 0.9999995$$

$$cos((1 + 4\pi)/1000) \approx 0.99991$$

Instead, the denominator should be equal to the estimated maximum value that x can take.

The number of collisions may be further reduced by applying multiple cosine functions with different periods. In that case, the encoding is a vector comprising the output of each cosine function. In other words, the output of every cosine function is concatenated into a vector which is the encoding.

Our approach relies on the encoding proposed by Vaswani et al. [43] which leverages an alternation of cosine and sine functions with an increasing denominator. More formally, the encoding of a numerical value x is a vector $\boldsymbol{pe}_x$ of dimension d whose j-th value is given either by equation 6.1 or 6.2 depending on whether j is even ($j = 2i$) or odd ($j = 2i + 1$), respectively.

$$pe_{x,2i} = sin(x/10000^{2i/d}) \tag{6.1}$$

$$pe_{x,2i+1} = cos(x/10000^{2i/d}) \tag{6.2}$$

As the authors underlined, there exists a linear relation between the $\boldsymbol{pe}_x$ and $\boldsymbol{pe}_{x+k}$, which they hypothesized should facilitate learning. Figure 6.5 illustrates the encoding.

6.3.3 Addition or Concatenation

Let us now investigate how to combine the arguments embedding and encoding into a single event representation. The two most common approaches are the addition and the concatenation.

One may describe the addition of two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ as the translation of a point $\boldsymbol{x}$ by a vector $\boldsymbol{y}$ – or equally a point $\boldsymbol{y}$ by a vector $\boldsymbol{x}$. Let us consider the system call name and

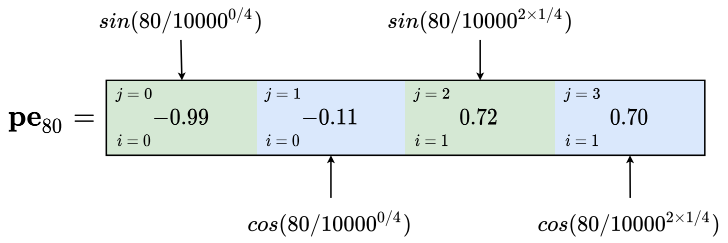


Figure 6.5 Encoding of the value $x = 80$ using Vaswani et al. [43] formula and a dimension $d = 4$.

the argument `entry` which has two possible values, “entry” and “exit”. Furthermore, let us consider the system call name embedding as a point and the `entry` embedding as a vector. The addition effectively shifts the system call name embedding depending on whether the event corresponds to the start or the end of the system call execution. As illustrated by figure 6.6, the relation has been explicitly modelled in the same space as the embedding of the system call name, which simplifies their visualization and interpretation.

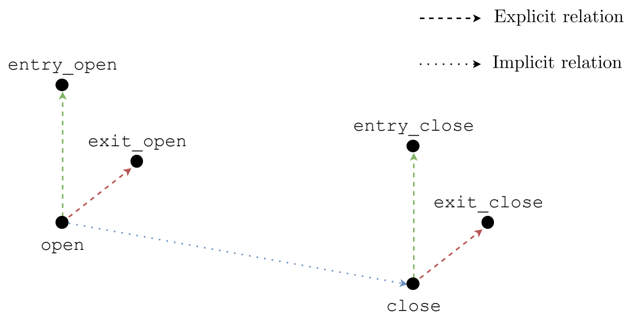


Figure 6.6 The embedding of the system call names “open” and “close”. The green and red dashed lines represent the explicitly modelled “entry” and “exit” relations, respectively. The blue dotted line represents the implicitly modelled relation between the two system call names.

The addition preserves the dimension, which may be too small to store all the information, thus creating a bottleneck. Instead, the concatenation allows combining vectors without such a bottleneck. Indeed, the dimension of the resulting vector is the sum of the dimensions of the concatenated vectors. That may, however, be a drawback if the model size scales with the input dimension as larger models are computationally expensive to train and prone to overfitting. One may mitigate the overfitting by collecting a sufficiently large dataset.

Although the embedding visualization is outside of the scope of this work, we believe interesting to model the system call name, the argument `entry`, and the argument `ret` in the same space. One may gain insights into the system by investigating the relations between those vectors. Therefore, only those values will be added, and the remaining arguments will be concatenated. Note that it would be ill-advised to add an encoding to an embedding since the former is not inherently meaningful.

6.3.4 Event Representation

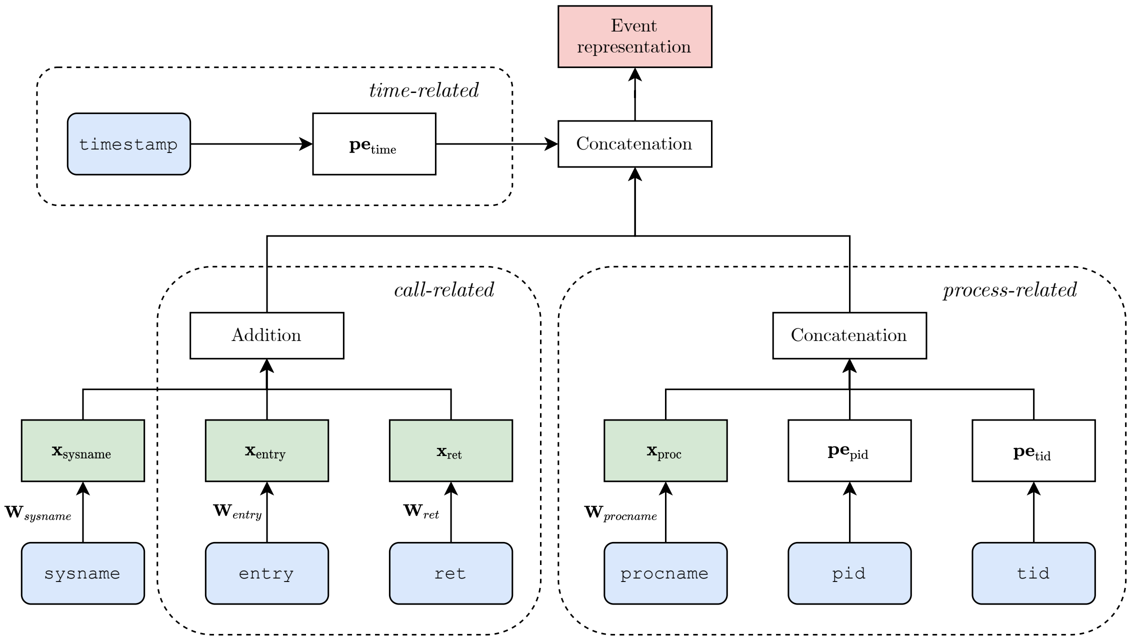


Figure 6.7 Computational graph of the event representation. Blue rounded rectangles represent the arguments. Green rectangles indicate that the transformation is learned (embedding), and the parametrization is noted next to the incoming arrow. White rectangle indicates that the transformation is not learned (encoding, addition, or concatenation).

Figure 6.7 illustrates the computational graph of the event representation. For the call-related arguments, the embedding of the `sysname`, the `entry`, and the `ret` are added. Note that the return value is simplified to either "success" if the numerical value is greater or equal to zero, or "failure" otherwise. For the process-related arguments, the `procname` embedding is concatenated with the `pid` and `tid` encodings. For the time-related argument, the `timestamp` is converted from nanoseconds to microseconds and is encoded. Finally, the representation of each category of arguments is concatenated.

Neural networks take numerical values as input that may be arranged as vectors, matrices, or, more generally, tensors. In the case of traces, the network's input is typically a sequence of vectors corresponding to the events. Such vectors may be the one-hot encoding of system call names, or better, their embedding. The proposed approach outputs a vectorial representation of the event with its arguments; therefore, it applies to most deep learning models.

The proposed event representation is non-contextual: a system call with its arguments will have the same representation regardless of the other trace events. Some tasks greatly benefit from a contextual representation which may be obtained with a Transformer trained on the masked language model objective [51]. Although such a model has been evaluated, contextual representations are outside the scope of this paper.

6.4 Data Collection

Over the years, many tracing datasets have been explored; however, most of them are not publicly available. Consequently, the now-obsolete UNM [182] and KDD98 [183] datasets are still widely used [13]. Those datasets were collected more than two decades ago and are clearly not representative of modern systems anymore. Therefore, they should not be used to evaluate recent approaches. In 2013, Murtaza et al. [13] and Creech and Hu [184] addressed this issue by introducing two new datasets: FirefoxDS and ADFA-LD, respectively. Unfortunately, the former is unavailable, and the system call arguments were omitted from the latter.

As indicated by Brown et al. [50], increasing the size of language models greatly improves their performance regardless of the task. As larger models require more data to be properly trained, the dataset must not only be modern but also massive. To the best of our knowledge, no massive and modern datasets comprising the system call arguments are publicly available. To that extent, we propose to generate such a dataset using *requests*. A request is a task delimited by specific start and end events. Examples include database queries, micro-services, and application functions. Notably, web requests have been extensively studied in the literature as they are ubiquitous. We introduce a methodology to generate a massive dataset of web request traces using a simple client-server framework (see Figure 6.8). The source code and the dataset are publicly available on GitHub[5] and Zenodo[6], respectively.

6.4.1 Methodology

On the client-side, a benchmark tool is used to send many concurrent requests to the server via the hypertext transfer protocol (HTTP). We chose `wrk2`[7], an open-source multithreaded equivalent of the Apache benchmark, as it guarantees a constant throughput load with an accuracy up to 99.9999% for sufficiently long runs. Moreover, `wrk2` yields a latency summary which allows extracting statistics about the dataset without processing it.

[5]`https://github.com/qfournier/syscall_args`

[6]`https://zenodo.org/record/4091287#.X4hhGNjpNQI`

[7]`https://github.com/giltene/wrk2`

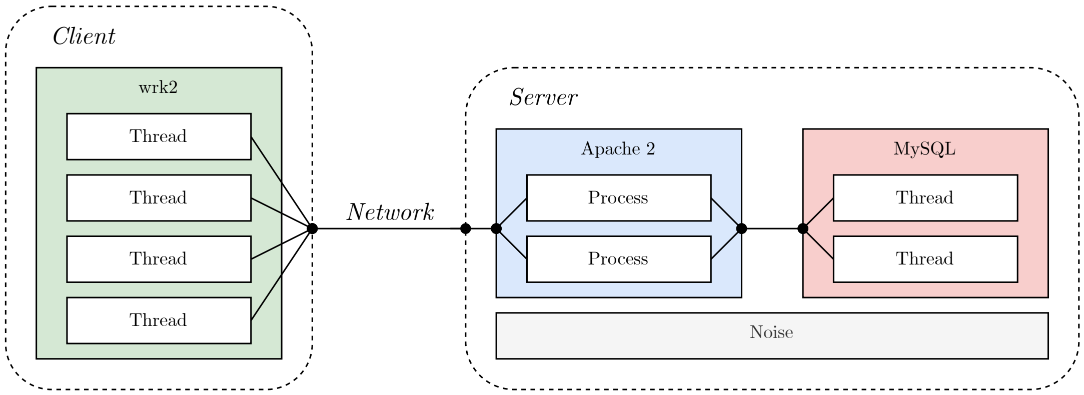


Figure 6.8 Client-server framework. The client and the server are two distinct physical machines that communicate over the network. The server may be executing other software while handling a request which is considered to be noise from a request point of view.

On the server-side, a web server handles the client requests and communicates with a database to retrieve the necessary information. For the web server, we chose Apache2 for its omnipresence and its modularity. Indeed, Apache2 is the most popular web server since 1996, and its vast community has developed many optional modules, including app servers and database connection managers. For the database, we chose MySQL for its ease of use and performance. MySQL is filled with the Sakila Sample Database[8] which includes an `author` table comprising ids, first names, and last names. Finally, PHP was installed as an Apache module to query the database.

One may be interested in simulating different behaviours such as slow or abnormal requests. In order to increase the likelihood of such requests, the server must be overloaded, which is done by restricting the amount or speed of the resources (CPU, memory, network, and disk). Consequently, Apache2 is deployed in a virtual machine using Virtual Box.

Physical servers often execute multiple tasks simultaneously. Since our server was dedicated, Firefox was automatically and randomly called from the console to take screenshots of random Wikipedia pages. The monitoring tools `htop` and `bmon` were also running in separate terminals. This allows creating a load on the CPU, the disk, and the network, as well as generating random events in the trace which adds variability.

In this work, we focus on the server-side since it is the source of most delays. A single trace is collected during the entire benchmark, therefore containing many individual requests. Depending on the task at hand, one may consider the whole trace as a single sequence or individual requests as separate sequences. Several tracers are available; however, the *Linux*

[8]`https://dev.mysql.com/doc/sakila`

Tracing Toolkit: next generation (LTTng) [10] is often the prefered choice given its lightweight and rapidity. Although only some system calls arguments are considered in this work, all arguments have been collected in order to have a complete view of the system.

6.4.2 Dataset Analysis

The server was deployed on a virtual machine with two cores from an Intel Core i7-8700 (up to 4.6 GHz), 1 Gb of DDR4 RAM, and an NVME SSD. The operating system was Ubuntu 18.04. Different throughputs were used to simulate different usages: idle, low, medium, and high. High usage means that the server is barely able to handle requests in real-time and that some end up with a timeout. Note that the training set and the test set were collected separately using different throughputs to avoid any overlap.

We collected around 250,000 requests which amount to almost 150 million system calls. One would likely have to collect a larger dataset in order to consider additional arguments such as the file descriptor without overfitting.

Figure 6.9 depicts the distribution of process names. As expected, the three most frequent processes are those that handle requests, namely the web server, its workers, and the database. Note that Firefox is responsible for issuing 13% of the system calls.

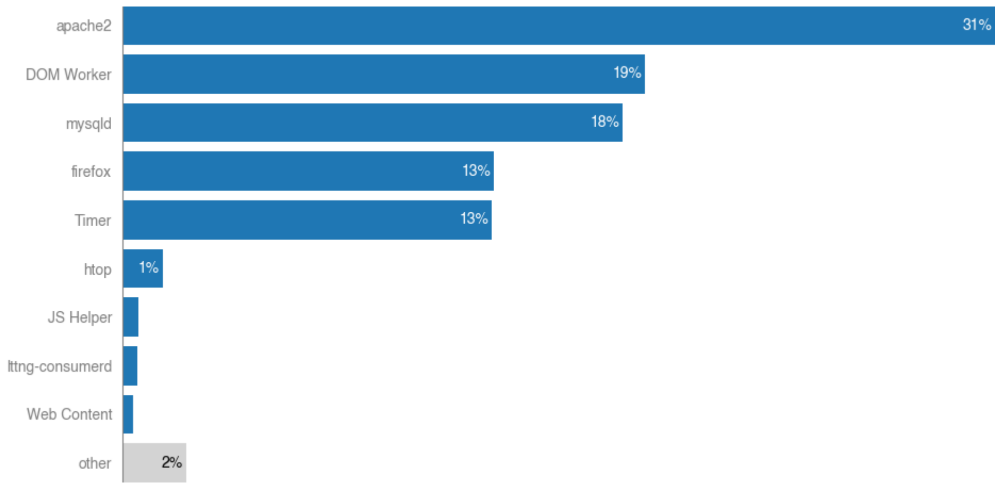


Figure 6.9 Distribution of process names.

Figure 6.10 depicts the distribution of system call names. The two most frequent system calls are `futex` and `poll` which provide a method for waiting until a condition becomes true and until a file descriptor becomes available to perform IO operations, respectively. This behaviour is to be expected in networked multicore systems, especially when many remote requests are being handled concurrently.

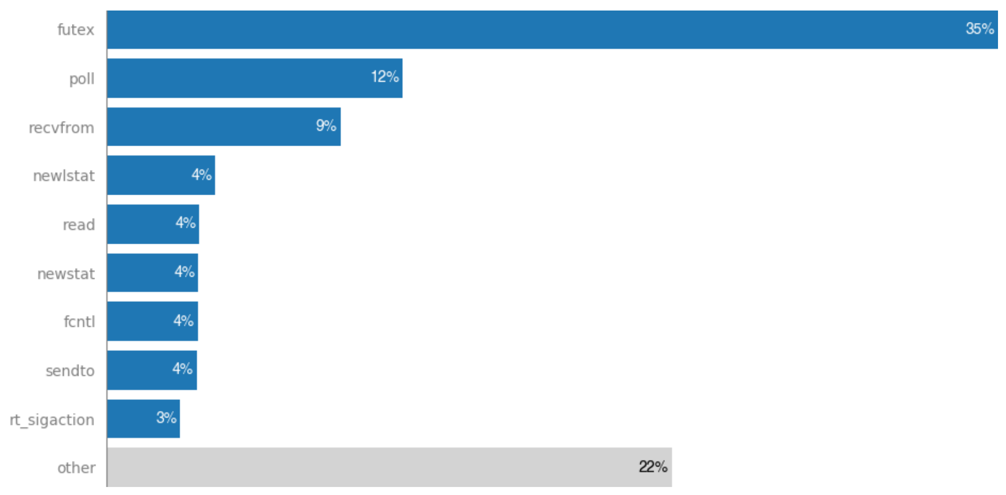


Figure 6.10 Distribution of system calls.

For an equivalent analysis of Ciena's dataset, we refer the reader to the GitHub repository.

6.5 Computational Experiments

This section introduces the neural networks and objectives on which the system call arguments' impact was evaluated. The source code, hyperparameters, and trained models are publicly available on GitHub[9].

6.5.1 Networks

The first model evaluated is a deep unidirectional Long Short-Term Memory (LSTM) network with two hidden layers comprising 96 units. The vast majority of existing works to analyze traces apply an LSTM on system call names only [36, 20, 2, 17, 19]; therefore, those methods would require almost no modification to leverage the arguments with the proposed approach.

The second model evaluated is a Transformer. Transformers are highly parallelizable and are able to learn dependencies across an unlimited number of steps at the price of quadratic complexity. Many works address this limitation; however, since this paper aims to demonstrate the usefulness of the system call arguments, we settled for the vanilla Transformer introduced by Vaswani et al. [43]. In particular, the network consists of six layers, each comprising 8 attention heads and a feedforward network with 128 units.

Contrary to LSTMs, Transformers are agnostic to the event position in the sequence. To solve this shortcoming, Vaswani et al. [43] injected positional knowledge by summing a positional

[9]`https://github.com/qfournier/syscall_args`

encoding with the embedding. In our experiments, the model achieved better results when the positional encoding was concatenated to the event embedding.

The dimensions of the arguments embedding and encoding have a significant impact on the model performance; thus, various configurations were evaluated. The following dimensions performed well in all experiments: 32 for the `sysname`, `entry`, and `ret`, 16 for the `procname`, 4 for the `pid` and `tid`, and 8 for the `timestamp`. Consequently, the dimension of the whole event representation is 64. Note that the dimension of the positional encoding was equal to that of the `timestamp`.

6.5.2 Objectives

The first objective is the left-to-right language model (LM), which predicts the conditional probability of the next system call name given the previous system calls. The chain rule allows computing the joint probability of the whole sequence, that is, its likelihood, and therefore may be used to detect changes in the system behaviour, intrusions, and anomalies. Notably, Kim et al. [2] used language modelling for host-based intrusion detection.

The second objective is the masked language model (MLM), which independently estimates the probability of masked words given the rest of the sequence. The more events are masked, the less context is available, and the more difficult is the training. In practice, MLM is often used to pre-train neural networks, and it has been shown to improve the model performance on downstream tasks, that is, the tasks of interest. Therefore, we evaluated the pre-trained model on LM in a zero-shot manner and determined that masking 25% of the events performed reasonably well on both datasets (see Table 6.5). In particular, we followed the methodology of Devlin et al. [51] by randomly selecting 25% of the events, of which 80% were entirely masked, 10% were replaced by a random system call name with the same argument values, and 10% were left unchanged. Randomly replacing the selected events generates noise which increases the robustness of the model. The proportion of random events is identical to Devlin et al. [51] as their ablation study showed it worked well for pre-training. Note that masked LMs are technically not language models as they are not trained to maximize the joint probability of sentences. Figure 6.11 illustrates the masked language model.

6.5.3 Data

Due to memory constraints on the graphics processing unit (GPU), the models must be trained on small sequences. Therefore, the traces were split into non-overlapping sequences of 256 events. Note that those sequences do not correspond to requests. One would need to

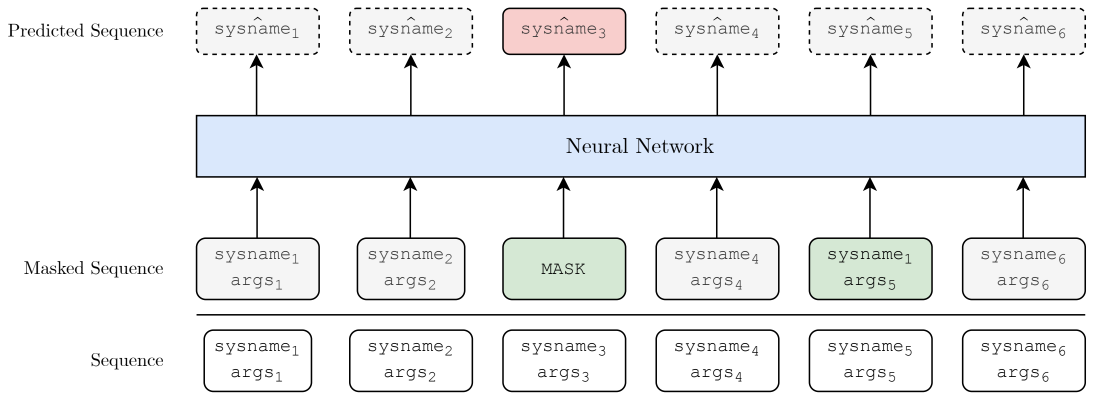

Predicted Sequence
$\hat{\text{sysname}}_1$
$\hat{\text{sysname}}_2$
$\hat{\text{sysname}}_3$
$\hat{\text{sysname}}_4$
$\hat{\text{sysname}}_5$
$\hat{\text{sysname}}_6$
Neural Network
Masked Sequence
sysname_1 args_1
sysname_2 args_2
MASK
sysname_4 args_4
sysname_1 args_5
sysname_6 args_6
Sequence
sysname_1 args_1
sysname_2 args_2
sysname_3 args_3
sysname_4 args_4
sysname_5 args_5
sysname_6 args_6

Figure 6.11 Masked language model. Events in green have been randomly selected, and system call names in red are the predictions independently considered.

implement the proposed approach with a lower computational complexity model in order to process whole requests as they usually contain thousands of events.

The first dataset studied has been introduced in Section 6.4 and comprises 318,674 training sequences and 258,190 test sequences. The second dataset has been collected by Ciena on pre-production servers executing proprietary software and comprises 190,924 training sequences and 64,628 test sequences. Although smaller, this dataset is designed to be representative of a real-world use case.

A quarter of each test set was randomly selected to create a validation set on which the hyperparameters were fine-tuned, and the model was evaluated at train time for early stopping.

6.5.4 Results

For each combination of datasets, objectives, and neural networks, two event representations have been compared: the system call name without any argument (`none`) and with every argument as described in Figure 6.7 (`all`).

Arguments may affect the performance differently; however, the computational cost of evaluating the impact of each argument, or worse, each combination of arguments, is prohibitive in practice. Instead, we evaluated the global impact of three groups of arguments: call-related (`entry` and `ret`), process-related (`procname`, `pid`, and `tid`), and time-related (`timestamp`).

Because the arguments embedding and encoding are concatenated, considering additional arguments increases the event representation dimension, which also increases the model size. On one side, the additional information allows the network to be larger without overfitting; hence one may see the increase in size as a byproduct of the arguments. On the other side, one may argue that a larger model only considering the system call name would perform better. In order to test these hypotheses, a compensated model considering no argument is evaluated (`none cmp.`). The dimension of the `sysname` embedding is increased from 32 to 64, which is the event representation's dimension when all the arguments are considered.

The model performance was measured in terms of cross-entropy (lower is better) and top-1 accuracy (higher is better). The cross-entropy is a measure of the difference between two distributions, in our case, the model output and the label. In the usual case of one-hot labels, the cross-entropy is defined as the negative logarithm of the correct event's predicted probability. The top-1 accuracy is the percentage of correct predictions, where a prediction is the system call name with the highest predicted probability. Results are detailed in Table 6.2.

Table 6.2 Impact of three categories of system call arguments (cross-entropy / accuracy).

			`none`	`none cmp.`	`time`	`call`	`process`	`all`
Web Requests	LM	LSTM	0.528 / 83.1	0.529 / 83.1	0.526 / 83.2	0.451 / 85.6	0.443 / 85.7	**0.423** / **86.4**
		Transformer	0.609 / 80.3	0.506 / 83.3	0.599 / 80.6	0.489 / 84.3	0.452 / 85.0	**0.380** / **87.3**
	MLM	Transformer	0.535 / 81.7	0.485 / 82.8	0.524 / 81.8	0.400 / 87.2	0.423 / 85.0	**0.182** / **94.1**
Ciena	LM	LSTM	0.294 / 91.8	0.301 / 91.5	0.301 / 91.6	0.277 / 92.2	0.283 / 91.9	**0.264** / **92.4**
		Transformer	0.323 / 90.4	0.292 / 91.3	0.310 / 90.8	0.290 / 91.5	0.271 / 91.9	**0.238** / **92.8**
	MLM	Transformer	0.285 / 90.8	0.264 / 91.3	0.270 / 91.2	0.202 / 94.0	0.245 / 91.8	**0.125** / **96.2**

In every experiment, the models that consider all the arguments achieved the lowest cross-entropy and the highest accuracy. The compensated models perform on par or better than their smaller counterpart; however, they are systematically outperformed by the models considering all the arguments. These results indicate that the increase in performance is not only due to the increase in model size but also to the additional arguments. Therefore, the arguments must contain useful information for language modelling tasks. Interestingly, the masked language model objective benefits more from call-related arguments than process-related ones.

The time-related argument has a negligible impact on the LSTMs; consequently, the temporality must be of little use for the left-to-right language model objective. Nonetheless, Transformers appear to benefit from the `timestamp` and, as a result, an ablation study of the `timestamp` and the `position` was conducted to quantify their impact. The results shown in table 6.3 reveal that `timestamp` does increase the performance over a model without any arguments, although not as much as the `position`, which indicates that Transformers are able to leverage the redundancy of the positional information embedded in the `timestamp`. However, with an equal number of parameters, a model considering only the `position` performs on par or better than one considering both values. Such behaviour is to be expected since the positional information in the timestamp is harder to extract. One may be tempted to dismiss the `timestamp`; however, it should be noted that some downstream tasks, including latency detection, may greatly benefit from the `timestamp`.

Table 6.3 Impact of the event's position and timestamp encoding dimensions on the Transformer without arguments (cross-entropy/accuracy). A dimension of zero is equivalent to omitting the argument.

`timestamp`	`position`	Web Requests	Ciena
0	0	0.730 / 76.9	0.444 / 86.9
8	0	0.661 / 78.4	0.337 / 89.8
0	8	0.609 / 80.3	0.323 / 90.4
8	8	0.599 / 80.6	**0.310** / **90.8**
0	16	**0.587** / **80.9**	0.313 / **90.8**

As shown in Table 6.4, the computational overhead imposed by the additional arguments was negligible compared to the overall training cost, making the proposed approach suitable for real-world applications. This is to be expected as the embedding is simply a matrix multiplication, and the encoding is only a small number of cosine and sine functions.

Table 6.4 Average epochs time (± std) in milliseconds of the Transformers trained on the web requests.

	LM	MLM
`none`	99.3 (± 2.0)	232.2 (± 8.2)
`none cmp.`	102.2 (± 1.6)	232.8 (± 5.3)
`time`	104.1 (± 2.6)	228.5 (± 3.8)
`call`	102.4 (± 2.1)	227.0 (± 6.6)
`process`	103.6 (± 1.7)	234.3 (± 6.9)
`all`	106.0 (± 2.4)	238.5 (± 4.5)

Table 6.5 Impact of the percentage of selected events for pre-training the Transformer with all arguments as evaluated on LM (cross-entropy/accuracy).

p_{mask}	Web Requests	Ciena
0.05	3.826 / 54.6	1.738 / 80.1
0.10	3.881 / 55.7	1.641 / 80.2
0.15	**3.314** / **56.7**	1.639 / 80.2
0.20	3.543 / 56.1	1.617 / **80.4**
0.25	3.334 / 56.1	1.647 / **80.4**
0.30	3.387 / 56.3	**1.548** / 80.2

6.6 Threats to Validity

The main threat to validity is the limited scope of the evaluation. Indeed, the approach has only been evaluated on two unsupervised language modelling tasks due to the lack of a publicly available dataset comprising the system call arguments. To mitigate this limitation, we provide the source code as well as the trained models for researchers and practitioners to evaluate our approach to their task.

The second threat to validity is the simplicity of the environment on which our dataset was collected. Consequently, the dataset may not represent real-world use cases and may not reflect the approach's actual benefit. This limitation is addressed by evaluating the two objectives on a second dataset collected by Ciena on pre-production servers. Additionally, our dataset is unlabelled. Consequently, it is challenging to use for supervised tasks such as anomaly detection. To alleviate this shortcoming, we provide a tutorial and the scripts required to generate the dataset such that users can produce their own labels.

Finally, although the proposed approach's computational overhead is negligible, neural networks still require powerful GPUs to be trained. The models' average training time described in Table 6.2 was less than 2 hours, with the slowest model taking about 5 hours on a single NVIDIA RTX2080Ti and two Intel Xeon Bronze 3104 1.7Ghz. Therefore, the experiments are easily reproducible with modest computational resources.

6.7 Concluding Remarks

In practice, it is often difficult to determine whether a specific deep learning approach is beneficial for the task at hand. In this section, we answer two general questions to help researchers and practitioners decide whether to adopt the proposed method.

Do the arguments invariably increase the model performance? We argue that the performance either improves or remains the same, provided two conditions. Firstly, the model must be flexible enough to be able to extract relevant information from the arguments. Such a model would be able to leverage the additional information in order to make more informed predictions, hence more accurate. If the arguments only contain irrelevant information to the task, the performance cannot increase. It may, however, decrease. Indeed, larger inputs translate into larger embeddings, which increase the model size, hence its flexibility. As the model flexibility increases, it becomes prone to overfitting, that is, to learn peculiarities from the dataset that do not reflect real explanatory factors. It is well-known that the difference between training and generalization errors grows with the model flexibility and shrinks with the number of training examples [28]. Therefore, the second condition is that enough samples must be available to prevent the model from overfitting. Large datasets of traces are typically easy to obtain, so the amount of data is not a limiting factor. Notably, this work introduced a methodology to generate a massive dataset of requests. Furthermore, many techniques such as dropout [124], batch normalization [108], and early stopping [185] allow mitigating the overfitting that may occur. Nonetheless, the arguments should be omitted if one knows beforehand that the information is irrelevant to the task. For instance, if a single thread is recorded, the `tid` is constant and may be safely omitted.

In practice, how does one know when to consider additional arguments? It seems that one would need to estimate a priori (1) if the model is complex enough, (2) if the dataset is large enough, and (3) if the arguments could be relevant to the task at hand. Fortunately, in the case of neural networks, the models are generally more flexible than necessary – they contain many more parameters than there are samples in the dataset [185]. As explained above, collecting large datasets of traces is often trivial, and the risk of overfitting may be significantly reduced. When possible, we recommend considering the arguments and comparing the model with a baseline that does not.

In this work, we introduced a massive dataset of web requests and a general approach to learning a representation of the system call names along with their arguments. By leveraging the left-out information, we were able to systematically increase the performance of two neural networks on two language-modelling tasks at a negligible computational cost. Possible future works include extending the embedding to userspace events, applying the models to downstream tasks such as anomaly detection, and applying the embedding to the many previous works that rely on LSTMs.

6.8 Acknowledgment

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena, and EffciOS for funding this project.

CHAPTER 7 ARTICLE 4: LANGUAGE MODELS FOR NOVELTY DETECTION IN KERNEL TRACES

Authors Quentin Fournier, Daniel Aloise, and Leandro R. Costa.

Submitted at Transactions on Modeling and Performance Evaluation of Computing Systems (12 October 2022)

Abstract Due to the complexity of modern computer systems, novel and unexpected behaviors frequently occur. Such deviations are either normal such as software updates and new users, or abnormal such as misconfigurations, latency, intrusions, and bugs. Regardless, novel behaviors are of great interest to developers, and there is a genuine need for efficient and effective methods to detect them. Nowadays, several researchers consider system calls to be the most fine-grained and accurate source of information to investigate the behavior of computer systems. Accordingly, this paper introduces a novelty detection methodology that relies on a probability distribution over sequences of system calls, which can be seen as a language model. Language models allow estimating the likelihood of sequences, and since novelties deviate from previously observed behaviors by definition, they would be unlikely under the model. Due to the success of neural networks, notably for language models, three architectures are evaluated in this work: the widespread LSTM, the state-of-the-art Transformer, and the lower-complexity Longformer. Large neural networks typically require a massive amount of data to be trained effectively, and to the best of our knowledge, no modern and massive datasets of kernel traces are publicly available. In order to address this limitation, this paper introduces a new open-source dataset of kernel traces comprising over 2 million web requests with seven distinct behaviors. The proposed methodology requires minimal expert hand-crafting and achieves an F-score and AuROC greater than 95% on most novelties while being data- and task-agnostic.

Keywords AIOps, Novelty Detection, Anomaly Detection, Deep Learning, Natural Language Processing, Language Models, LSTM, Transformer.

7.1 Introduction

Despite the fact that computer systems are virtually deterministic, complex interactions between hardware and software often result in novel and unexpected behaviors. Novel behaviors correspond to any deviation from what has been previously observed and include common behaviors such as component upgrades, software updates, new users, and rare queries, as well as anomalies such as misconfigurations, latency, intrusions, hardware failures, and bugs. This research focuses on detecting novelties rather than anomalies since it is a broader problem and since normal yet novel behaviors such as significant changes in user habits are often of interest to practitioners. Moreover, anomaly detection methods may be ineffective in detecting clever attacks designed to resemble legitimate users, which may still be detected as novel behaviors.

A non-intrusive and lightweight approach to recording the behavior of computer systems is to trace them. Tracing is the act of collecting low-level events generated whenever a specific instruction called tracepoint is encountered at runtime. This research considers events generated by the operating system called kernel events since they expose the behavior of the whole system [186]. Furthermore, kernel events allow tracing virtually any Linux system without modifying the source code since tracepoints are already implemented in the Linux kernel. In particular, this paper focuses on a subset of the kernel events named *system calls* or syscall. System calls correspond to requests from applications running in the userspace to the kernel in order to access resources such as memory, network, or other devices that would otherwise be inaccessible. In short, system calls are the only way for an application to communicate with the operating system. As mentioned by Kim et al. [2], many researchers consider system calls to be the most fine-grained and accurate source of information to analyze computer systems.

Due to the computational speed of modern computers, operating systems often execute hundreds of system calls every second, making the manual analysis of a collection of kernel traces with tools such as Trace Compass[1] too time-consuming. As a result, practitioners and researchers often analyze traces automatically [178]. Since novel behaviors are unexpected and unknown by definition, their detection is difficult to specify in practice. Consequently, novel behaviors are typically detected with machine learning techniques since they learn to solve the task from examples [14, 34, 35]. Nonetheless, most machine learning algorithms benefit from or require carefully hand-crafted features [13, 27]. For the past decade, the majority of research has focused on neural networks [2, 20, 17] since they automatically learn to extract meaningful features for the task, thereby reducing the need for an expert and ultimately improving the performance. Since traces are sequences of discrete values comprising a syntax and a semantic akin to natural languages [16], deep learning techniques from natural language processing (NLP) are particularly well suited for traces.

[1]`https://www.eclipse.org/tracecompass/`

Natural language processing is the use of natural language by a computer and includes various tasks such as question answering, machine translation, summarization, sentiment analysis, and image captioning. A wide range of NLP applications rely on a probability distribution over sequences of tokens, often words or characters, called a language model (LM) [28]. One of the most popular approaches to learning a language model is the left-to-right LM, whose objective is to predict the conditional probability of each token knowing the previous ones. Formally, given a sequence of N tokens $\boldsymbol{w} = \{w_1, w_2, \ldots, w_N\}$, the left-to-right language model computes for each token w_i the conditional probability $P(w_i|w_{i-1}, \ldots, w_1)$. The chain rule of probability states that the joint probability of the entire sequence is the product of all the conditional probabilities:

$$P(w_1, w_2, \ldots, w_N) = \prod_{i=1}^{N} P(w_i|w_{i-1}, \ldots, w_1) \tag{7.1}$$

Neural network language models typically minimize the cross-entropy loss, which is equivalent to maximizing the joint probability of the sequence. In other words, such language models maximize the likelihood of the known behaviors. By definition, novel behaviors deviate from what has been previously observed. Therefore, they have a low likelihood under a language model trained on known behaviors. The proposed approach is novelty-agnostic, that is, suitable for detecting any deviations from previously observed behavior, including misconfigurations, bugs, intrusions, and latency. Moreover, the approach is data-agnostic, that is, suitable for a wide range of data, including userspace traces, kernel traces, and logs.

Previous research studied log and trace language models for anomaly and novelty detection [2, 4, 3, 5]. Our approach improves over and diverges from existing approaches in three critical points: (1) the quality and quantity of the data are drastically improved, (2) neural networks that are able to learn extremely long dependencies are investigated, and (3) the novelty detection methodology takes into account the sequence length. We next expand on these three contributions.

First, deep learning approaches are known to greatly depend on the quality and quantity of the data [28]. Nonetheless, the public datasets considered by current research such as UNM [182], KDD98 [183], and ADFA-LD [184] are small and obsolete as explained by Creech and Hu [184] and Murtaza et al. [13]. Furthermore, these datasets lack the system call arguments, a valuable piece of information that has been shown to improve the performance of neural networks for language models [16]. As a result, these datasets are inadequate for effectively training large neural networks and are not representative of modern systems. As a solution, this paper introduces a massive dataset of kernel traces that includes system call arguments and comprises more than 2 million web requests from seven realistic scenarios, including misconfigurations and latencies. Notably, our dataset enables training larger neural networks, such as the Transformer, that have become state-of-the-art in other fields. The dataset has been made public, as well as the data collection methodology and the scripts for reproducibility.

Second, current anomaly and novelty detection approaches rely on recurrent neural networks (RNNs), most often the Long Short-Term Memory [38] (LSTM) network. However, recurrent networks are unable to efficiently model long-term dependencies due to their iterative nature. As explained by Khandelwal et al. [49], LSTM language models sharply distinguish recent positions but only vaguely remember the distant past. Dai et al. [48] estimated that the relative effective context length (RECL) of LSTMs on natural language is between 200 and 400 tokens, which is consistent with Khandelwal et al. [49] estimation. This inherent limitation of recurrent networks is analyzed in the case of kernel traces since they are typically much longer. In particular, this paper investigates the state-of-the-art network for sequence processing called the Transformer [43], whose main advantage is the ability to model dependencies of arbitrary length. However, this flexibility comes at the expense of a quadratic complexity with respect to the sequence length. Consequently, a linear-complexity alternative called the Longformer [142] is also investigated.

Third, anomaly and novelty detections are typically performed with a top-k on the individual conditional probabilities [3, 17, 5] or a threshold on the joint probability of the sequence [2, 4]. However, conditional probabilities are only able to detect deviations of single events, also known as point outliers, while the joint probability does not take into account the sequence length. As a solution, our methodology leverages the *perplexity*, a prevalent measure of how well a probability model predicts a sample [43, 51].

The remainder of this paper is organized as follows. Section 7.2 surveys the related works. Section 7.3 introduces the proposed novelty detection methodology. Section 7.4 presents the dataset collection methodology and analyzes the dataset. Section 7.5 details the experiments and reports the results of the proposed approach on the collected dataset. Section 7.6 acknowledges the threats to internal and external validity. Section 7.7 discusses the strengths and limitations of the proposed methodologies as well as suggests interesting future research avenues. Finally, Section 7.8 concludes this paper.

7.2 Related Work

Let us preface the related works by discussing the distinction between anomaly and novelty. As defined in Section 7.1, a novelty is any deviation from previously observed behaviors. Novelties include anomalies since they are typically unknown and unexpected, but not all novelties are anomalies. For instance, new users and rare queries are novel yet normal behaviors. The vast majority of the literature focuses on anomalies, and one of the most popular approaches is learning a "normal" behavior from the data and identifying any deviations from this behavior as abnormal [2, 3, 4, 5, 6]. We argue that these approaches would be better framed as novelty detection methods as an additional mechanism would be necessary to determine whether the novel behaviors are normal or abnormal. For that reason, even though this paper focuses on novelty detection, most of the approaches discussed in this section were published under the anomaly detection paradigm.

This section surveys the fundamental aspects of the relevant related works: (1) the trace representation, (2) the machine learning model, and (3) the anomaly or novelty detection scheme.

7.2.1 Trace Representation

A trace usually comprises millions of low-level events, each containing multiple arguments, making them resource-intensive to handle. As a result, researchers have traded information for compactness in three ways: reducing the number of arguments, aggregating the events across time, and extracting higher-level features.

The first and foremost approach is to reduce the number of arguments. Current research often exclusively considers the event names and ignores the arguments, such as the process name and the return value. However, arguments are valuable data that allow the model to make more informed and, ultimately, more accurate predictions. Indeed, temporal information such as the response time [17, 18], the timestamp [16], and the duration [3] has recently been considered with great success. Instead of reducing the number of arguments, Ezeme et al. [19] compressed the values of the arguments by encoding the characters using ASCII values and considering the frequency distribution of these values for each argument. Nonetheless, contemporary research demonstrated the benefit of considering the actual values of multiple system call arguments for neural language models [16].

The second approach is to aggregate the events across time. The main example of this approach is called bag-of-words, also known as system call counts vector [20], frequency counts of system call names [21] or bag of system calls [22]. A bag-of-words is a representation that describes the number of occurrences of each token within a document. For instance, consider a vocabulary $\mathbb{V} = \{a, b, c\}$ and a sequence $\boldsymbol{w} = \{a, c, c, c, a, c\}$. The bag-of-word representation of this sequence is $[2, 0, 4]$. The aggregation trades the ordering and fine-grained temporal information for a more compact representation. However, temporal information may be critical to detecting some novelties, such as latency.

The third approach is to extract higher-level features from the trace, such as states of kernel modules [13] or execution states [27]. Although carefully hand-crafted higher-level features may deliver excellent performance, they discard the fine-grained information that makes traces so valuable. Moreover, they are time-consuming, error-prone, and potentially suboptimal since they must often be hand-crafted specifically for the task considered.

7.2.2 Machine Learning Model

Due to the widespread use of computer systems and the importance of detecting anomalies and novel behaviors, a wide range of machine learning techniques have been explored, including rule-based algorithms [14, 15], naive Bayes [34, 35], decision trees [36, 35], hidden Markov models [24, 35], and support vector machines (SVM) [35]. Given the great success of deep learning, researchers have recently shifted toward a family of neural networks called recurrent neural networks (RNNs) [37].

Recurrent neural networks iteratively process variable-size sequences by sharing the parameters at each position, as illustrated in Figure 7.1. They have been successfully applied to a wide range of applications, including speech recognition [187, 188], image captioning [189], machine translation [40], and anomaly detection [2, 3]. RNNs have the advantage of iteratively storing information into their memory, also referred to as hidden representation, allowing information from prior input tokens to influence the current output. In other words, the RNN output depends on the previous tokens within the sequence. More formally, let us consider a recurrent neural network that processes the i-th token x_i from the input sequence $\boldsymbol{x} = \{x_1, x_2, x_3, \ldots, x_N\}$. The model builds a hidden representation h_i as a function of the current token x_i and the previous hidden representation h_{i-1}, which encapsulates information from all previous tokens. The model then produces the i-th output y_i as a function of the hidden representation h_i and proceeds to the next token x_{i+1}.

Due to vanishing and exploding gradient issues, vanilla RNNs are unable to learn long-term dependencies [190]. As a solution, Hochreiter and Schmidhuber [38] introduced the now widely popular long short-term memory (LSTM) network, which mitigates these shortcomings with paths through time. Alternatively, Cho et al. [39] introduced the gated recurrent unit (GRU), which behaves and performs similarly to the LSTM while requiring fewer pa-

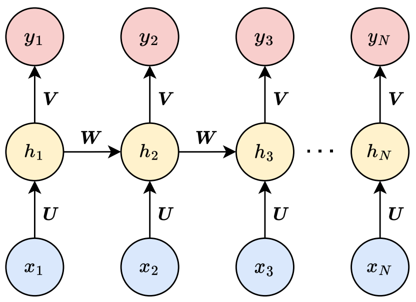


Figure 7.1 The unrolled computational graph of a recurrent neural network. There is a one-to-one mapping between the input sequence $\boldsymbol{x} = \{x_1, x_2, x_3, \ldots, x_N\}$ and the corresponding output sequence $\boldsymbol{y} = \{y_1, y_2, y_3, \ldots, y_N\}$. The weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$, and $\boldsymbol{W}$ are shared across all positions.

rameters.

The LSTM and GRU have been at the core of numerous anomaly and novelty detection approaches. For instance, Kim et al. [2] detected host-based intrusions with an ensemble of LSTMs trained on sequences of system call names. Dymshits et al. [20] identified changes in the behavior of processes with unidirectional and bidirectional LSTMs trained on sequences of bag-of-words. Song et al. [36] identified and explained anomalies with an LSTM trained on time-series data obtained from traces. Nedelkoski et al. [17] detected anomalies with a multimodal network made of the concatenation of the hidden representations of two LSTMs trained on textual logs and real-valued response time. Lv et al. [26] detected intrusions by extending sequences with a GRU in a sequence-to-sequence fashion.

Recurrent neural networks process sequences iteratively, as illustrated by Figure 7.1. As a result, RNNs suffer from memory compression [41] and are unable to model very long-term dependencies. In particular, LSTM language models only vaguely remember the distant past [49] and are unable to model dependencies that span more than 200 to 400 tokens [49, 48]. One solution is to augment the LSTM with an attention mechanism. Ezeme et al. [19] and Brown et al. [4] detected anomalies with LSTMs augmented with inter-attention. However, their inherently sequential nature prevents parallelizing them.

As of the writing of this research, recurrent neural networks, including the more complex LSTM and GRU, have been surpassed and frequently replaced by the Transformer [43]. As illustrated by Figure 7.2, the Transformer is a simple network architecture based solely on two attention mechanisms: the inter-attention and the self-attention. In particular, self-attention enables relating tokens in the input sequence and, as such, replaces the role of recurrences in RNNs. Specifically, the self-attention mechanism computes a pairwise compatibility score between tokens corresponding to how much each token contributes to each output. Consequently, the Transformer is able to model arbitrary length dependencies. Furthermore, since this compatibility score is computed independently for each pair of tokens, the Transformer processes entire sequences simultaneously and can be efficiently parallelized. However, these major benefits come at the cost of quadratic complexity with respect to the sequence length. Consequently, a plethora of efficient alternatives have been proposed, such as the Longformer [142], the Linformer [88], the Reformer [87], and Big Bird [64]

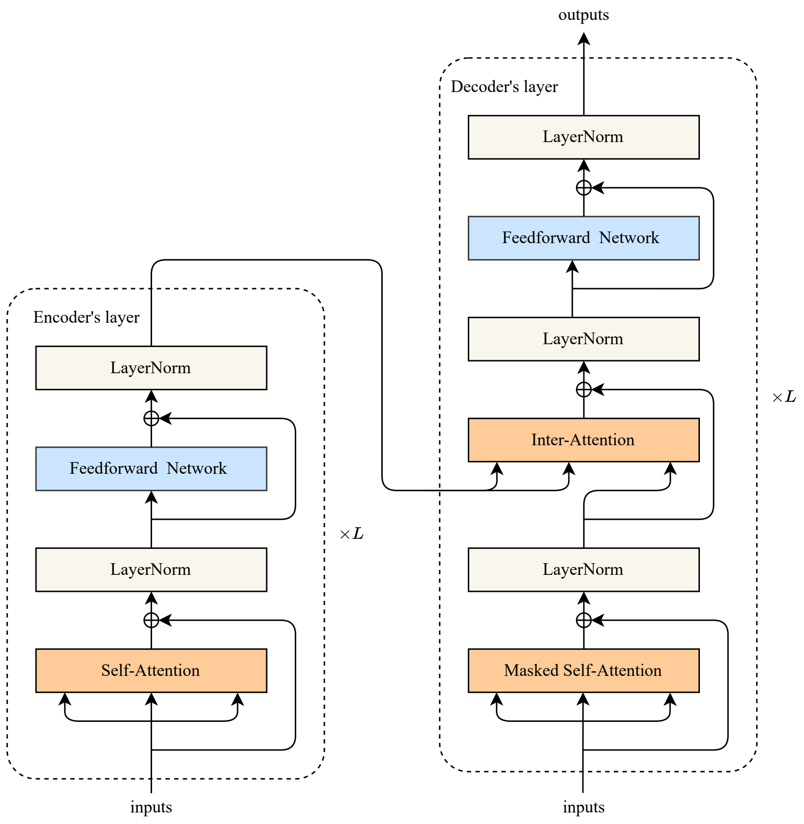


Figure 7.2 The computational graph of the Transformer, which comprises an encoder and a decoder. The encoder first processes the entire input sequence and produces a representation of each token attended by the decoder to generate the output sequence in an autoregressive manner.

Despite the clear advantages and successes of the Transformer, researchers have not yet investigated this architecture to detect novelties in traces. Nonetheless, the Transformer has been considered for related tasks such as the evaluation of the system call embedding proposed in [16] and for related data such as logs anomaly detection [6, 5]. Note that there is a clear distinction between logs and traces since the former are significantly higher-level than the latter, requiring distinct analyses with unique challenges. However, to the best of our knowledge, no lower-complexity Transformer has been applied to detect novelties or anomalies in traces or logs. For complementary information on the attention mechanisms and efficient Transformers, we refer the reader to the many extensive and comprehensive surveys that have been published [46, 47, 70, 191].

7.2.3 Detection Scheme

Real-world datasets are often unlabeled regarding anomalies since they are typically unknown beforehand. Besides, manually labeling them afterward is generally time-consuming and error-prone. Consequently, the vast majority of approaches fall into the self-supervised or unsupervised setting. This paper focuses on language models such as the left-to-right LM that outputs the conditional probability of every possible token for each token in the sequence. In the literature, there are two distinct detection schemes based on the idea that novelties correspond to mispredictions.

The first scheme assumes that a misprediction occurs when the correct token does not appear in the top-k most likely predictions. Guo et al. [6] considered a sequence as anomalous if it contains more than a certain number of mispredictions, whereas Bogatinovski et al. [5] considered the ratio of mispredictions. Temporal information is decisive in detecting some novelties, such as latencies. Consequently, in addition to the token mispredictions, Du et al. [3] and Nedelkoski et al. [17] predicted the timestamp and the response time, respectively.

The second scheme considers that a misprediction occurs when the conditional or joint probability is lower than a given threshold. Notably, Kim et al. [2] and Brown et al. [4] detected anomalies with a threshold on the negative log-likelihood of the whole sequence.

7.3 Methodology to Detect Novelties With Neural Language Models

This section introduces the proposed methodology and focuses on the same fundamental aspects as the literature review: (1) the trace representation, (2) the machine learning model, and (3) the novelty detection scheme.

7.3.1 Trace Representation

Neural networks learn to extract the relevant features for a task and thus typically benefit from richer inputs. Accordingly, this paper follows the methodology proposed in [16] and relies on a joint representation of the system call name (`sysname`), the timestamp (`timestamp`), and five context fields that are added to all system calls by LTTng, namely the return value (`ret`), the process name (`procname`), the thread id (`tid`), the process id (`pid`), and whether the event corresponds to the start or the end of a system call execution (`entry`).

To determine how to represent the arguments, one must first identify the inherently meaningful ones – whose values convey meaning in themselves without any context. For explanatory purposes, let us consider a system call whose process name is `mysql` and whose process id is `15371`. The process name indicates that a MySQL database emitted the call, while the process id does not provide knowledge but allows relating the events emitted by the same process in the context of the trace. Out of the considered arguments, the `sysname`, `ret`, `entry`, and `procname` are inherently meaningful, while the `tid`, `pid`, and `timestamp` are not[2].

The semantic knowledge contained in the values of the inherently meaningful arguments is encapsulated in a compact vectorial representation called embedding. An embedding effectively acts as a lookup table, as illustrated by Figure 7.3, and is defined by a dense matrix $\boldsymbol{W} \in \mathbb{R}^{d_v \times d_e}$ where d_v is the size of the vocabulary and d_e is an hyperparameter corresponding to the dimension of the embedding such that $d_e \ll d_v$.

$$\underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}}_{\substack{\text{One-hot vector indicating} \\ \text{the token's position in the vocabulary}}} \times \underbrace{\begin{bmatrix} 4.3 & 1.1 & 4.9 \\ 2.1 & 8.6 & 8.5 \\ 6.8 & 7.5 & 0.1 \\ 1.9 & 1.0 & 3.0 \\ 2.2 & 1.1 & 8.5 \\ 6.2 & 3.6 & 9.7 \end{bmatrix}}_{\text{Embedding matrix } \boldsymbol{W}} = \underbrace{\begin{bmatrix} 2.2 & 1.1 & 8.5 \end{bmatrix}}_{\text{Embedding}}$$

Figure 7.3 The embedding of the fifth element in a vocabulary comprising $d_v = 6$ elements. The embedding has a dimension $d_e = 3$ and is hence a more compact representation of the token.

The values of the context-dependent arguments, such as the `pid` or `tid`, could be directly provided to the network as they are numerical values. However, it is best practice to normalize the input to mitigate potential numerical instabilities and speed up training [192]. As a result, the context-dependent arguments are encoded with a succession of cosine and sine functions, as proposed by Vaswani et al. [43]. Formally, the encoding of a numerical value x is a vector

[2]There are exceptions, such as `pid` 0 and 1, which are meaningful.

$\boldsymbol{pe}_x$ of dimension d computed as follows:

$$pe_{x,i} = \begin{cases} \sin\left(x/1000000^{i/d}\right) & \text{if } i \text{ is even,} \\ \cos\left(x/1000000^{i-1/d}\right) & \text{otherwise,} \end{cases} \tag{7.2}$$

where d is a hyperparameter.

In order to produce a joint representation of the system calls with their arguments and to provide a single input to the network, the embeddings and encodings must be combined. As mentioned in [16], the addition requires the vectors to have the same dimension and preserves that dimension, which may be too small to store all the information, thus creating a bottleneck. Consequently, the concatenation of the embeddings and encodings vectors is preferred, as shown in Figure 7.4.

Our methodology diverges from that of [16] and [43] in three aspects. First, the timestamps are converted into elapsed times between two consecutive system calls to avoid numerical instabilities as they exceed the largest value that can be stored on 32 bits. Indeed, POSIX timestamps are defined as the number of nanoseconds since January 1, 1970. Second, the denominator of the encoding is increased from 10^4 to 10^6 (see Equation 7.2), as the values encoded are larger than in the work of Vaswani et al. [43]. Finally, the embeddings and encodings are all concatenated since this empirically resulted in more effective models for our data and task.

7.3.2 Neural Networks

The proposed methodology was evaluated on a simple n-gram baseline, the widespread LSTM, the state-of-the-art Transformer, and the lower-complexity Longformer. Let us briefly introduce and justify each method.

The n-gram model makes the Markov assumption that the conditional probability may be approximated by only considering the $n-1$ tokens instead of all previous tokens. In other words, the n-gram model approximates the conditional probability $P(w_i|w_{i-1}, w_{i-2}, \dots, w_1)$ as $P(w_i|w_{i-1}, w_{i-2}, \dots, w_{i-(n-1)})$, which is computed in practice as the number of times that $\{w_{i-1}, w_{i-2}, \dots, w_{i-(n-1)}\}$ is followed by w_i out of all the occurrences of $\{w_{i-1}, w_{i-2}, \dots, w_{i-(n-1)}\}$ in the dataset.

Following the vast majority of literature, the proposed methodology was evaluated using a unidirectional multi-layer LSTM. Since this architecture is well known and ubiquitous in the literature, the reader is referred to the original paper by Hochreiter and Schmidhuber [38]

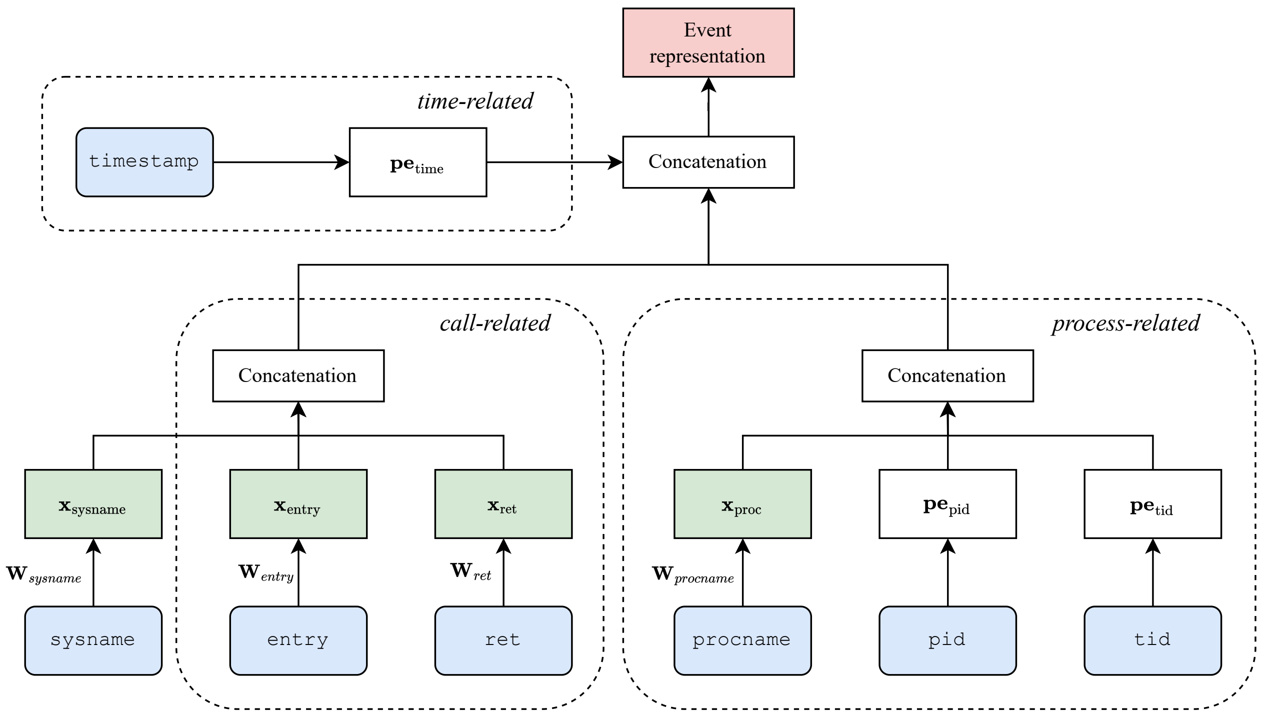


Figure 7.4 The computational graph of the system call representation. The rounded blue rectangles represent the considered arguments. The green rectangles represent the learned transformations (embedding), for which the parametrization is annotated next to the incoming arrow. The white rectangle transformations are not learned (encoding and concatenation).

and the reference book of Goodfellow et al. [28] for a comprehensive description and analysis of the model.

However, as discussed in the introduction, kernel traces are typically much longer than the effective context length of LSTMs. As such, they may contain dependencies that the LSTM is unable to model. In order to investigate this potential limitation, the proposed methodology was evaluated on a vanilla Transformer.

The Transformer is a model that is able to process variable-length sequences without recurrences, relying instead on the self-attention mechanism. As said in Section 7.2, the Transformer has quickly become state-of-the-art for sequence processing due to its capacity to model arbitrary length dependencies. However, the flexibility of the Transformer comes at the cost of a quadratic complexity with respect to the sequence length. In practice, even with multiple GPUs, mixed precision, and gradient checkpointing, the Transformer is unable to handle entire kernel traces. Consequently, sequences were truncated. As a result, the model was only able to compute an estimate of the joint probability. In order to determine whether truncating sequences is a potential issue for kernel traces and to propose a solution that is more easily deployable in practice, a lower-complexity Transformer called the Longformer [142] was selected based on the attention patterns learned by the Transformer.

The quadratic complexity of the Transformer originates from the computation of the pairwise compatibility score between each pair of tokens. As a solution, the Longformer achieves a linear complexity by replacing the self-attention mechanism with a combination of two sparse attention mechanisms called global tokens and sliding windows. Since the objective of the left-to-right language model is to predict the next token given the previous ones, future tokens are masked to prevent the model from looking ahead at the solution. Instead of looking at all previous tokens, the sliding window attention only considers the past k tokens, similar to the n-gram model. Since only a fixed number of positions are considered for each token, the complexity of the window attention is linear with respect to the sequence length. Additionally, the global tokens are able to attend to every position and be attended by every position. Given that there is a fixed number of global tokens, each considering every token in the sequence, the complexity of global tokens is also linear with respect to the sequence length. Figure 7.5 depicts the full attention of the original Transformer and the sparse attention mechanisms of the Longformer. For complementary information, the reader is referred to the comprehensive surveys published on efficient Transformers [70, 191].

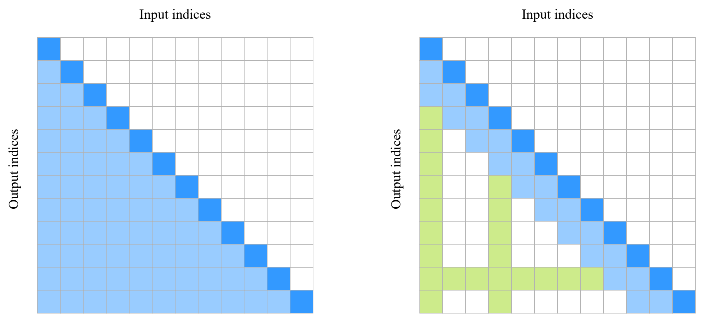


Figure 7.5 (*Left*) The connectivity matrix of the Transformer's full attention. (*Right*) The connectivity matrix of the Longformer's sparse attention. The window attention and global tokens are depicted in blue and green, respectively. The i-th output position attends to the j-th input position if, and only if, the cell (i, j) is colored. The diagonal is highlighted to ease the reading. The matrices are lower triangular as future positions are masked to prevent the model from looking ahead at the solution.

7.3.3 Novelty Detection Scheme

The proposed methodology assumes that the distribution of system calls encapsulates the behavior of the system and that deviations from the known distribution correspond to novelties. As explained in Section 7.1, a language model is a probability distribution over sequences of tokens. Consequently, the neural networks are trained with the left-to-right language model, whose task is to predict the next token knowing the previous ones.

Given an input sequence $\boldsymbol{w} = \{w_1, w_2, \ldots, w_N\}$ comprising N tokens from a vocabulary $\mathbb{V}$, a neural left-to-right language model outputs the conditional probability $P(w^*|w_{i-1}, \ldots, w_1)$ for each w^* in the vocabulary and for each position $i = 1, \ldots, N$, such that $\sum_{w^* \in \mathbb{V}} P(w^*|w_{i-1}, \ldots, w_1) = 1$. The joint probability $P(w_1, w_2, \ldots, w_N)$ is given by the chain rule of probability as $P(w_1, w_2, \ldots, w_N) = \prod_{i=1}^{N} P(w_i|w_{i-1}, \ldots, w_1)$. Figure 7.6 illustrates the neural left-to-right language model with a toy sequence $\boldsymbol{w} = \{a, c, c, a\}$ and vocabulary $\{a, b, c\}$.

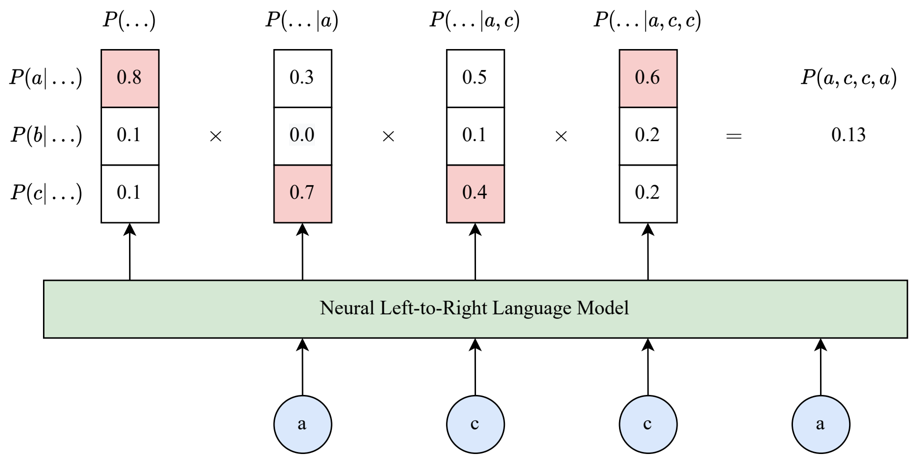


Figure 7.6 Neural left-to-right language model.

However, since the conditional probabilities are lower than 1 in practice, longer sequences are more likely to have a lower joint probability. Let us consider an operating system that produces 200 system calls per second and two requests of 80 ms and 120 ms. Let us assume that the variation in response time is normal and that the events are all equally likely, with a conditional probability of 95%. The two requests comprise $200 \times 0.08 = 16$ and $200 \times 0.12 = 24$ system calls, respectively. Consequently, their likelihood is $0.95^{16} = 44\%$ and $0.95^{24} = 29\%$, respectively. As a result, the likelihood of sequences is not well suited for novelty detection, as the throughput of system calls is high, and the sequence lengths may greatly vary.

The *perplexity* of a language model is a widely popular metric [43, 51] that measures its degree of uncertainty when a new token is generated, averaged over very long sequences. In order to describe the perplexity, let us first look at the per-word entropy H of a sequence of word $\{w_1, w_2, \ldots, w_N\}$ generated by a language model:

$$H = \lim_{N\to\infty} -\frac{1}{N} \sum_{w_1,w_2,\ldots,w_N} P(w_1, w_2, \ldots, w_N) \log_2 P(w_1, w_2, \ldots, w_N) \tag{7.3}$$

Assuming ergodicity, in other words, that a sufficiently large set of samples generated by a random process is able to represent the average statistical properties of the entire process, the summation may be discarded:

$$H = \lim_{N\to\infty} -\frac{1}{N} \log_2 P(w_1, w_2, \ldots, w_N) \tag{7.4}$$

Given a large enough value of N, the entropy can be approximated as:

$$\hat{H} = -\frac{1}{N} \log_2 P(w_1, w_2, \ldots, w_N) \tag{7.5}$$

Finally, the perplexity is defined as:

$$PP = 2^{\hat{H}} = P(w_1, w_2, \ldots, w_N)^{-1/N} \tag{7.6}$$

where N is the sequence length. In the above example, the perplexities of both requests are equal to $0.95^{-\frac{16}{16}} = 0.95^{-\frac{24}{24}} = 1.05$.

A sequence with a higher perplexity than the sequences in the training set is less likely under the model and can therefore be detected as a novelty. In practice, a simple threshold is efficient and provides excellent results. The threshold is empirically determined for each novel behavior with the in-distribution and out-of-distribution datasets to maximize the F-score.

7.4 Data Collection

Real-world traces are seldom released due to security and privacy concerns. Consequently, researchers often rely on the UNM [182] and KDD98 [183] datasets. Nonetheless, these two datasets are more than twenty years old and thus fail to represent modern systems [184, 13]. As a solution, Creech and Hu [184] introduced ADFA-LD for host-based intrusion detection. However, ADFA-LD comprises only a few thousand samples without the system call arguments, which is too small for training large neural networks. Alternatively, Murtaza et al. [13] introduced the much larger FirefoxDS dataset. Unfortunately, FirefoxDS is no longer available at the time of writing.

Neural networks, especially neural language models, greatly benefit from scaling, as revealed by the current race toward ever-larger models [50, 193]. However, large neural networks greatly benefit from massive datasets [194], and to the best of our knowledge, no modern and massive datasets of kernel traces are publicly available. In order to address this limitation, this paper introduces a novel open-source dataset of kernel traces comprising over 2 million web requests with seven distinct behaviors. The dataset includes all the system calls arguments, and the requests are well delimited by userspace events and labeled according to their behavior.

The remainder of this section explains the data collection methodology in detail and analyzes the collected dataset.

7.4.1 Methodology

Similar to the methodology of [16], a benchmark tool sends numerous concurrent requests from the client to the server via the hypertext transfer protocol (HTTP). A web server receives the requests and calls PHP to query an SQL database and create the requested dynamic web page. The simple client-server architecture is depicted in Figure 7.7.

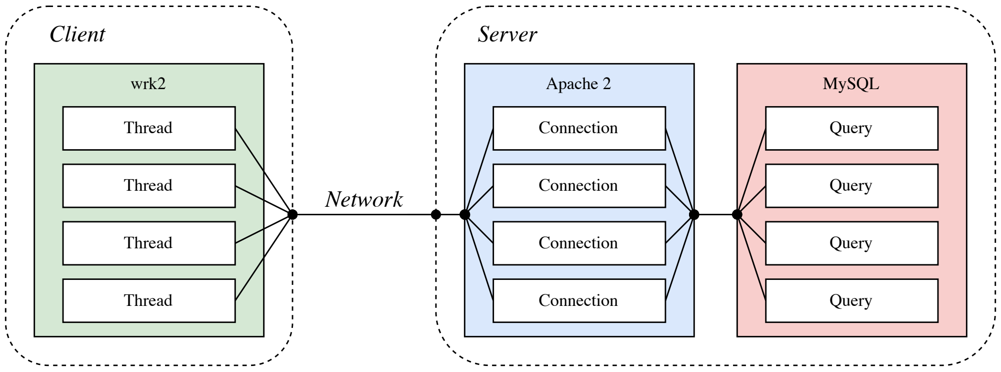


Figure 7.7 The client-server architecture. The client sends numerous concurrent HTTP requests over the network to the server. The web server queries an SQL database for each request to build a dynamic web page.

Let us briefly describe and justify the choice of software for the client-server architecture. On the client side, the requests were emitted with the wrk2[3] benchmark tool as it is an open-source and multithreaded alternative to the Apache benchmark that guarantees a stable throughput for sufficiently long execution times. On the server side, the requests were handled with the Apache2[4] web server as it is widely used in modern systems thanks to its modular design. Note that Apache2 was manually instrumented with two user-space events

[3]`https://github.com/giltene/wrk2`
[4]`https://httpd.apache.org`

`httpd:enter_event_handler` and `httpd:exit_event_handler` that delimit each request. The requested dynamic web pages were created by querying MySQL[5] with PHP installed as an Apache2 module. MySQL was chosen since it is an open-source relational database management system commonly used with Apache2. Finally, the database was filled with the Sakila sample database[6], as it is intended to provide a standard schema that can be used across numerous examples. Notably, this database comprises an author table with unique ids, first names, and last names.

Since developers rarely have access to the client side, this paper focuses on the server side, from where most novelties, such as latency and misconfigurations, indeed originate. The system calls as well as the user-space events that delimit each request were collected on the server with the Linux Trace Toolkit: next generation (LTTng) [10] due to its lightweight and rapidity [9].

Since the proposed methodology aims at detecting novelties with a language model, a set of known behaviors referred to as in-distribution (ID) must be available, as well as sets of novelties referred to as out-of-distribution (OOD). Accordingly, three in-distribution sets were collected with a typical configuration under nominal load (train ID, validation ID, and test ID), and two out-of-distribution sets were collected for each of the server-side novelties (validation OOD, test OOD). We describe next the six novel behaviors considered in this work:

- CPU: The workload generator tool stress-ng[7] overloads the CPU by performing numerous matrix multiplications. This behavior simulates a compute-intensive process competing for resources with the web server, which may arise from a cryptocurrency-mining procedure deployed by an intruder, for instance.

- OPcache: The server is misconfigured by disabling PHP's OPcache, which stores precompiled script bytecode in memory to speed up the response time of requests. Simply put, PHP compiles and stores in memory the binary of scripts without executing them such that the scripts do not have to be recompiled every time they must be executed. This behavior may arise from a developer who disabled the cache during development and forgot to enable it afterwards.

- Dump IO: The highest level of Apache2 log is enabled and stored into a log file with the dump_io mod[8]. Such detailed information is valuable for investigating the server

[5]`https://dev.mysql.com`

[6]`https://dev.mysql.com/doc/sakila/en/`

[7]`https://github.com/ColinIanKing/stress-ng`

[8]`https://httpd.apache.org/docs/2.4/en/mod/mod_dumpio.html`

behavior during debugging but requires writing enormous log files. As a result, this novel behavior heavily uses storage resources and lead to longer request response times. Similar to OPcache, this behavior may arise when a developer enables logging during development, but forgets to disable it afterwards.

- Connection: By default, Apache2 is configured to support 150 concurrent connections. However, it will start dropping requests as the traffic increases and the number of maximum concurrent connections is not increased. This behavior was reproduced by reducing the number of concurrent connections instead of increasing the traffic, as the server would be IO-bounded before requiring more connections. Besides, this decision allows the traffic to remain consistent with the other behaviors.
- Socket: With Apache2 KeepAlive enabled, the web server and browsers agree to reuse the same socket to transfer multiple files, thereby reducing the CPU usage at the cost of higher memory usage. By default, KeepAlive is enabled as CPU usage is typically the main limiting factor. For this out-of-distribution behavior, KeepAlive is disabled, which may arise when the Apache2 is redeployed from a memory-limited machine to a CPU-limited one.
- SSL: The secure sockets layer (SSL) is a protocol for establishing secure connections between the web server and browsers. For this novel behavior, SSL is disabled, which is a significant security issue due to misconfiguration.

7.4.2 Dataset Analysis

The Apache2 web server was deployed on an Ubuntu 21.04 machine equipped with 16-core Intel E5640 (up to 2.67 GHz) and 192 Gb of RAM. In order to load the server properly, the client relies on the wrk2 benchmark tool to perform 1000 requests per second.

The web server was traced for 1,000s for the in-distribution training set and 100s for the in-distribution and out-of-distribution validation and test sets. The datasets are balanced since there are as many positive samples (i.e., out-of-distribution or novel) as negative ones (i.e., in-distribution or known). However, in real-world applications, novelties are rare. Consequently, the number of positive samples is expected to be much smaller than that of negative samples, which may affect the evaluation. This decision is explained in the threat to validity section.

In this work, requests comprise all the system calls generated between their start and end as delimited by the userspace events `httpd:enter_event_handler` and `httpd:exit_event_handler` regardless of their thread ids (see Figure 7.8). Consequently, system calls may be added to multiple requests since they are concurrent. The primary reason behind this decision is that we want requests to include the events associated with the root cause of the novelties. For instance, let us assume that an unexpected process is taking CPU time, thus creating latency. To detect the root cause of this behavior, the events generated by this abnormal process must be included in the request, even though they do not have the same thread id. The main drawback of this approach is that requests contain significantly more events, making them more resource-intensive to process and increasing the noise.

```
Start Request A -> [13:59:47.282364969] (+0.000001277) server httpd:enter_event_handler: {cpu_id = 11},
                   {tid = 116291, pid = 116274}, …

                   [13:59:47.282366056] (+0.000001087) server syscall_exit_poll: {cpu_id = 14},
                   {procname = "php-fpm", pid = 116475, tid = 116475}, …

                   [13:59:47.282369674] (+0.000003618) server syscall_entry_recvfrom: {cpu_id = 14},
                   {procname = "php-fpm", pid = 116475, tid = 116475}, …

                   [...]

                   [13:59:47.282666723] (+0.000001116) server syscall_exit_futex: {cpu_id = 5},
                   {procname = "httpd", pid = 114788, tid = 114842}, …

Start Request B -> [13:59:47.282668855] (+0.000002132) server httpd:enter_event_handler: {cpu_id = 1},
                   {tid = 114847, pid = 114788}, {connection_state = 1}

                   [13:59:47.282669044] (+0.000000189) server syscall_entry_writev: {cpu_id = 11},
                   {procname = "httpd", pid = 116274, tid = 116291}, …

Start Request C -> [13:59:47.282670140] (+0.000001096) server httpd:enter_event_handler: {cpu_id = 5},
                   {tid = 114842, pid = 114788}, {connection_state = 1}

                   [13:59:47.282681550] (+0.000011410) server syscall_exit_writev: {cpu_id = 11},
                   {procname = "httpd", pid = 116274, tid = 116291}, …

                   [...]

                   [13:59:47.284127210] (+0.000000642) server syscall_exit_epoll_ctl: {cpu_id = 1},
                   {procname = "httpd", pid = 114788, tid = 114847}, …

End Request B ->   [13:59:47.284127270] (+0.000000060) server httpd:exit_event_handler: {cpu_id = 1},
                   {tid = 114847, pid = 114788}, …

                   [...]

                   [13:59:47.284666725] (+0.000000365) server syscall_exit_sendto: {cpu_id = 0},
                   {procname = "php-fpm", pid = 116475, tid = 116475}, …

End Request A ->   [13:59:47.284669780] (+0.000003055) server httpd:exit_event_handler: {cpu_id = 12},
                   {tid = 116291, pid = 116274}, …
```

Figure 7.8 Sample from a toy trace collected on a multicore server. Since the requests are defined to comprise all the system calls generated between their start and end, as delimited by two userspace events, request A includes requests B and part of request C.

Table 7.1 reports statistics on the requests of each set after discarding the first second, which corresponds to the initialization of LTTng. The significantly lower number of requests for the CPU behavior is due to the server's inability to maintain the throughput due to the lack of CPU.

Table 7.1 Statistics on the requests in each dataset.

Behavior	**Dataset**	**Number of Requests**	**Request Length**			**Request Duration** (ms)		
			min	mean	max	min	mean	max
ID	Train	999,063	238	1105.7 ± 244.8	4,645	0.28	1.68 ± 0.65	53.61
	Validation	99,058	30	1107.3 ± 244.9	2,803	0.03	1.67 ± 0.59	11.69
	Test	99,065	240	1108.7 ± 247.1	2,683	0.91	1.67 ± 0.61	12.36
Connection	Validation	99,016	246	1125.7 ± 243.0	2,882	0.94	1.66 ± 0.60	15.02
	Test	99,019	158	1125.0 ± 243.3	2,792	0.27	1.66 ± 0.60	11.83
CPU	Validation	57,616	258	1910.6 ± 607.6	6,221	1.31	13.25 ± 6.06	52.10
	Test	56,191	222	1913.8 ± 596.0	6,363	0.51	13.58 ± 5.81	35.69
IO	Validation	98,974	350	1827.7 ± 323.4	6,155	1.27	2.13 ± 3.23	349.33
	Test	98,980	392	1821.1 ± 321.0	6,967	1.25	2.10 ± 1.23	103.69
OPcache	Validation	99,069	256	1162.9 ± 244.2	2,824	0.99	1.78 ± 0.60	14.79
	Test	99,057	250	1160.6 ± 245.9	2,896	0.96	1.77 ± 0.60	11.94
Socket	Validation	99,074	216	2082.0 ± 362.5	8,463	0.83	6.89 ± 0.73	48.79
	Test	99,084	679	2081.8 ± 355.7	7,032	3.63	6.89 ± 0.64	19.61
SSL	Validation	99,072	16	1058.1 ± 229.1	3,230	0.04	1.48 ± 0.36	15.92
	Test	99,067	238	1054.8 ± 230.2	3,855	0.80	1.47 ± 0.38	22.23

Figures 7.9 and 7.10 display the distributions of system call names and process names, respectively, for the in-distribution training set and OPcache out-of-distribution validation set. Interestingly, both datasets present similar distributions in which the most frequent system calls are `recvfrom` and `read`, and the most common processes are `php-fpm`, `httpd`, `connection`, and `mysqld`. As a result, a simple model of the system call names and process names distributions would not be able to distinguish the two behaviors. Nonetheless, as we will present in the next section, our methodology achieves an F-score greater than 98% on that novelty. The readers are referred to the GitHub repository[9] for equivalent and additional analysis on all studied datasets.

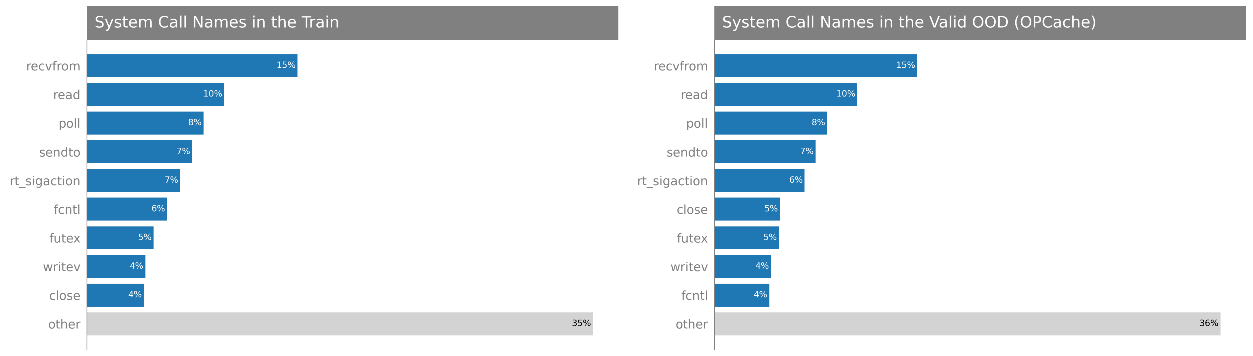


Figure 7.9 Distributions of system calls names in the in-distribution training set and OPcache out-of-distribution validation set.

[9] `https://github.com/qfournier/syscall_novelty_detection`

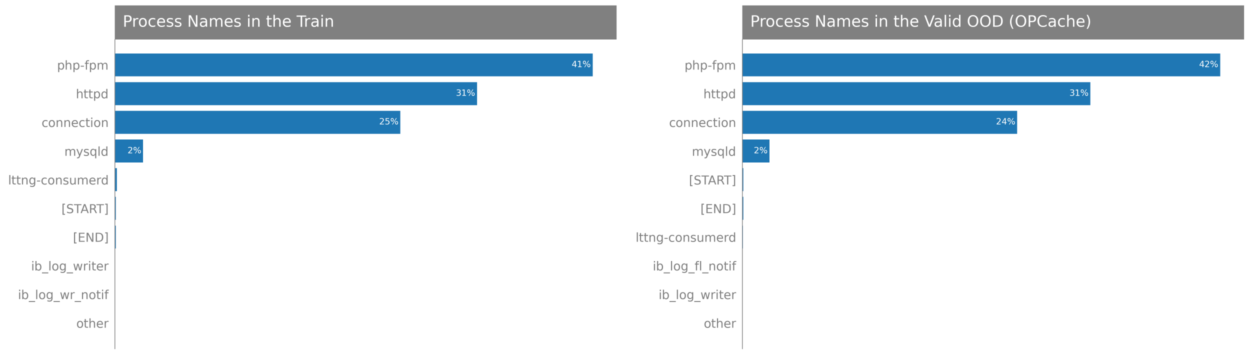


Figure 7.10 Distributions of process names in the in-distribution training set and OPcache out-of-distribution validation set.

7.5 Results

7.5.1 Language Models

The experiments were conducted on a modern server with 2 x Intel Gold 6148 Skylake @ 2.4 GHz, 4 x Nvidia V100SXM2 (16G memory), and 64G of memory. The source code and logs are publicly available on GitHub[10].

Due to the simplicity of implementing NVIDIA's Automatic Mixed-Precision, all the neural networks were trained with mixed precision, accelerating training and reducing memory consumption by storing and computing the weights, activations, and gradients in half-precision [100].

Due to the quadratic complexity of the Transformer with respect to the sequence length, Transformers were trained with gradient checkpointing, which trades memory for computation by recomputing the activations during the backward pass instead of storing them in memory during the forward pass [82]. Additionally, the few sequences longer than 2048 system calls were truncated to avoid exceeding the memory available.

Due to the computational cost and environmental impact of extensively tuning the hyperparameters with a grid search or a random search [195], the hyperparameters were manually selected in a greedy fashion. The hyperparameters considered for the three networks are the depth and width of the models, the embedding size, the optimizer, the warmup steps, the label smoothing weight, the dropout probability, the number of updates without improvements before reducing the learning rate, and the number of updates before early stopping. Additionally, the number of heads, the SwiGLU activation function [196], and the T-fixup

[10]`https://github.com/qfournier/syscall_novelty_detection`

initialization [115] were considered for the Transformer and the Longformer. Furthermore, the window size, the dilation, and the number of global tokens were also considered for the latter. More than 80 distinct neural network configurations were evaluated in total, each requiring more than a day of computation on the server described above. The exhaustive list of hyperparameters for each model is available on GitHub.

In order to learn a language model of the system calls, the networks were optimized with the left-to-right LM objective. Simply put, the networks predict the name of each system call given the previous system calls in each sequence and their arguments. Since the left-to-right LM objective is a multi-class classification problem, the loss function $\mathcal{L}(x, y)$ minimized is the cross-entropy defined as follows:

$$\mathcal{L}(x, y) = -\frac{1}{N} \sum_{n=1}^{N} \log \frac{\exp x_{n,y_n}}{\sum_{i=1}^{C} \exp x_{n,i}} \tag{7.7}$$

where N denotes the numbers of tokens in the sequence, C denotes the numbers of classes or system call names in the vocabulary, $x_{n,c}$ denotes the unnormalized conditional probability predicted by the model, and y_n denotes the true class or the index of the correct system call name. In addition to the cross-entropy, the top-1 accuracy is also reported. This work defines accuracy as the fraction of correctly predicted tokens. In other words, the accuracy corresponds to the number of times the most probable system call name corresponds to the actual one, divided by the total number of predictions.

Table 7.2 reports the average cross-entropy and top-1 accuracy of the three neural networks on the in-distribution and out-of-distribution sets. Each experiment was reproduced five times with different seeds to mitigate the stochasticity. The cross-entropy on the in-distribution sets is consistently and significantly lower than on the out-of-distribution sets, indicating that the networks have a higher degree of uncertainty when modeling the novel behaviors. The LSTM outperforms the two attention-based networks in terms of cross-entropy and accuracy, although they have the advantage of learning arbitrary length dependencies. As expected, increasing the width and depth of the two attention-based models significantly improved their performance in terms of cross-entropy and accuracy. For instance, increasing the depth from 2 to 6 layers and the width from 672 to 896 allowed reducing the cross-entropy of the Transformer from 0.907 to 0.719 on the in-distribution test set, outperforming the LSTM. However, although larger models were better at generalizing due to their higher flexibility, they performed poorly on our downstream novelty detection task since they assigned a high likelihood to all behaviors. Since our goal is to detect novelties, only the smaller models are reported in Table 7.2 and thereafter.

Table 7.2 Training Performance of the Neural Networks.

	LSTM		**Transformer**		**Longformer**	
Dataset	Cross-Entropy	Accuracy	Cross-Entropy	Accuracy	Cross-Entropy	Accuracy
Train	**0.714 ± 0.002**	**0.764 ± 0.000**	0.891 ± 0.039	0.701 ± 0.013	0.875 ± 0.008	0.712 ± 0.003
Test ID	**0.720 ± 0.002**	**0.762 ± 0.000**	0.907 ± 0.038	0.696 ± 0.012	0.885 ± 0.010	0.708 ± 0.004
Test OOD (Connection)	**0.812 ± 0.017**	**0.737 ± 0.006**	1.274 ± 0.103	0.605 ± 0.025	1.105 ± 0.018	0.651 ± 0.006
Test OOD (CPU)	0.961 ± 0.027	0.736 ± 0.010	1.155 ± 0.056	0.685 ± 0.012	**0.940 ± 0.022**	**0.744 ± 0.005**
Test OOD (IO)	2.287 ± 0.185	0.366 ± 0.037	2.993 ± 0.307	0.232 ± 0.042	**2.082 ± 0.150**	**0.391 ± 0.036**
Test OOD (OPcache)	**1.127 ± 0.019**	**0.669 ± 0.005**	1.302 ± 0.052	0.607 ± 0.013	1.254 ± 0.024	0.630 ± 0.007
Test OOD (Socket)	**1.008 ± 0.033**	**0.699 ± 0.007**	1.573 ± 0.138	0.549 ± 0.029	1.223 ± 0.033	0.636 ± 0.012
Test OOD (SSL)	**0.906 ± 0.018**	**0.716 ± 0.007**	1.495 ± 0.105	0.550 ± 0.030	1.245 ± 0.027	0.619 ± 0.010

Once trained, the language models allow estimating the perplexity of the requests. Figure 7.11, 7.12, and 7.13 depict the distribution of the perplexity of the request in the in-distribution test set and out-of-distribution OPcache test set computed with the LSTM. Although the distributions of the length and duration of the request are similar, the networks assign a higher perplexity to the out-of-distribution requests, indicating that the networks are able to leverage complex interactions between system calls instead of relying on simple metrics such as the length or duration of the requests. Equivalent figures for all datasets are available on GitHub.

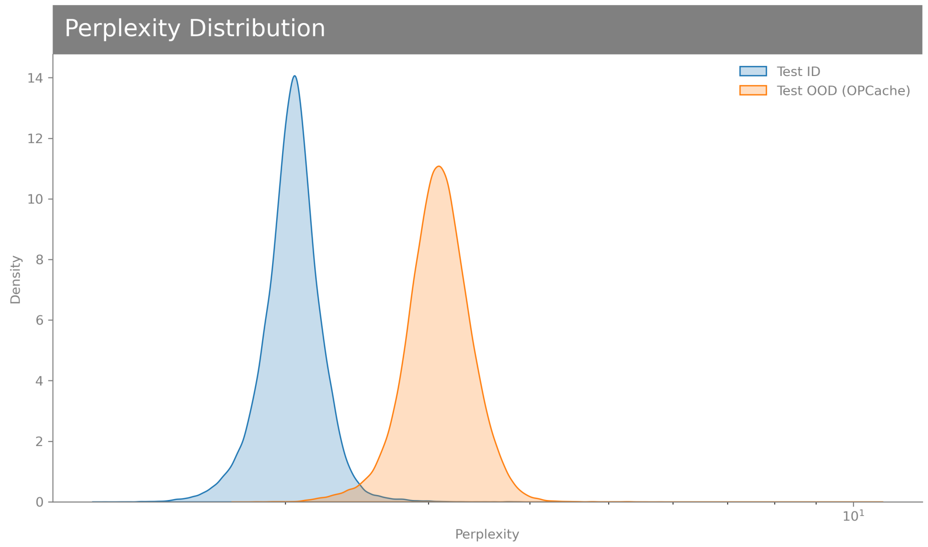


Figure 7.11 Distribution of the perplexity of the request in the in-distribution (blue) and out-of-distribution OPcache (orange) test sets computed with the LSTM.

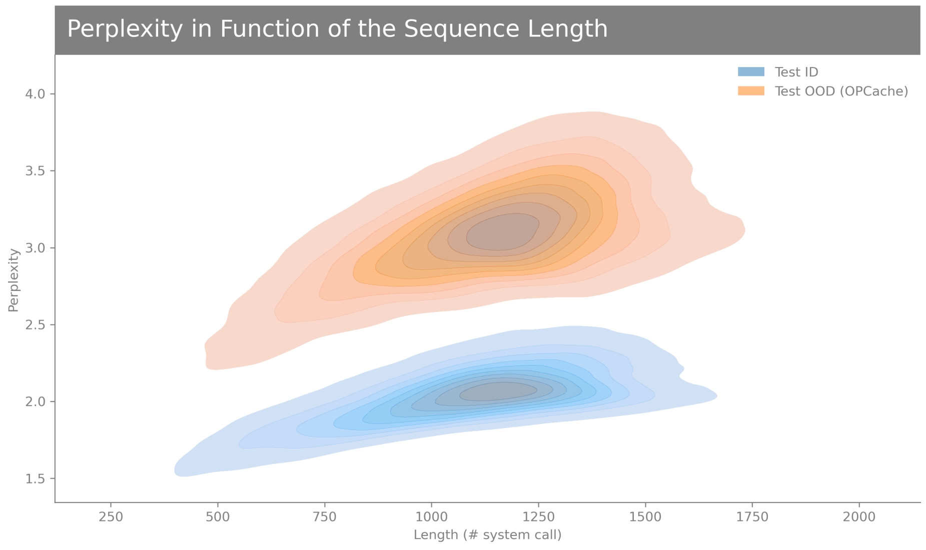


Figure 7.12 Distribution of the perplexity of the request in the in-distribution (blue) and out-of-distribution OPcache (orange) test sets with respect to the request length computed with the LSTM.

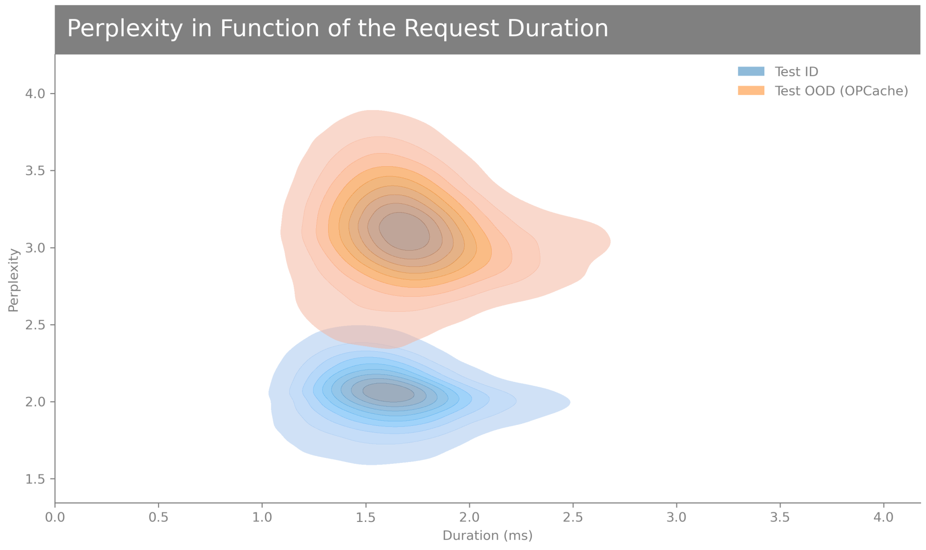


Figure 7.13 Distribution of the perplexity of the request in the in-distribution (blue) and out-of-distribution OPcache (orange) test sets with respect to the request duration computed with the LSTM.

7.5.2 Novelty Detection

The novelty detection task is a binary classification problem as requests are either in-distribution or out-of-distribution. The two outcomes of binary classification problems are referred to as *positive* and *negative*, with the former corresponding to the class of interest by convention. Thus, the positive class corresponds to the novelties or out-of-distribution sequences. A binary classifier may successfully predict novel behaviors as positives or known behaviors as negatives, such cases are called *true positives* (TP) and *true negatives* (TN), respectively, or misclassify novel behaviors as negatives or known behaviors as positives, the so-called *false negative* (FN) and *false positive* (FP), respectively. Binary classifiers are most often evaluated in terms of precision and recall, with the precision defined as the fraction of actual positives among the predicted positives and the recall defined as the fraction of actual positives correctly predicted:

$$\text{precision} = \frac{TP}{TP + FP} \quad \text{and} \quad \text{recall} = \frac{TP}{TP + FN} \tag{7.8}$$

In order to have a single measure, the harmonic mean of these values, denoted F-score or F-measure, is instead reported:

$$\text{F-score} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{7.9}$$

As explained in Section 7.3.3, the classification is performed with a simple threshold on the perplexity, which acts as a novelty score. In practice, the threshold that maximizes the F-score is empirically determined for each validation set. Then, the F-score is computed for each test set with that threshold.

Instead of selecting a threshold to maximize a given metric, the receiver operating characteristic (ROC) curve evaluates the ratio of true positives against the ratio of false positives at various thresholds values. In order to have a single measure, the area under the ROC curve (AuROC) is computed. The reader is referred to Zou et al. [197] for additional information on the ROC curve.

Table 7.3 reports the AuROC and F-score of the 4-gram baseline and the three neural networks. As we can observe, the simple baseline was unable to detect the novel behaviors accurately, with the surprising exception of the IO behavior. The simplicity of detecting this behavior arise from the out-of-distribution requests having a wildly different distribution of system call names compared to the in-distribution request. Indeed, the two most common system calls in the training set are `recvfrom` (15%) and `read` (10%), while they are `write` (17%) and `getpid` (17%) for the IO dataset.

Due to the recent successes of the Transformer over the LSTM, one would have expected the former to outperform the latter. However, that is not the case: the LSTM performed on par or better than the attention-based networks in terms of AuROC and F-score on 3 out of

Table 7.3 Novelty Detection Performance of the Language Models.

	4-gram		LSTM		Transformer		Longformer	
Dataset	AuROC	F-score	AuROC	F-score	AuROC	F-score	AuROC	F-score
Test OOD (Connection)	51.5	66.7	79.9 ± 3.8	74.8 ± 2.9	**97.8 ± 1.6**	**94.3 ± 2.9**	93.6 ± 1.4	87.6 ± 1.8
Test OOD (CPU)	0.9	53.2	**98.5 ± 0.6**	**93.6 ± 2.1**	94.8 ± 1.8	85.1 ± 3.4	67.9 ± 7.7	59.6 ± 4.1
Test OOD (IO)	98.6	94.7	**100.0 ± 0.0**	**100.0 ± 0.0**	**100.0 ± 0.0**	**100.0 ± 0.0**	**100.0 ± 0.0**	**100.0 ± 0.0**
Test OOD (OPcache)	65.2	67.5	**99.7 ± 0.1**	**98.3 ± 0.2**	99.1 ± 0.2	96.7 ± 0.4	98.9 ± 0.2	96.2 ± 0.4
Test OOD (Socket)	22.6	66.7	98.8 ± 0.6	94.7 ± 2.3	**99.9 ± 0.1**	**99.1 ± 0.6**	99.1 ± 0.4	96.4 ± 1.3
Test OOD (SSL)	50.5	66.7	91.9 ± 1.7	85.4 ± 2.0	**99.7 ± 0.2**	**98.5 ± 0.9**	98.4 ± 0.5	94.5 ± 0.9

6 behaviors. Figure 7.14 illustrates the attention activation patterns of the Transformer on the in-distribution validation set, which corresponds to the known and expected behavior. Interestingly, the dependencies modeled by the Transformer are mostly local since most of the attention learnt is along the diagonal. This observation justifies the choice of the less complex Longformer which relies on the window attention mechanism, thus assuming local dependencies. Furthermore, this observation explains the performance of the LSTM since RNNs are inherently biased toward local dependencies due to their iterative nature, making them well suited for this use case. Nonetheless, in cases where longer-term dependencies must be modeled or when a wide range of behaviors must be learned, we expect the Transformer or its lower-complexity alternative to perform better.

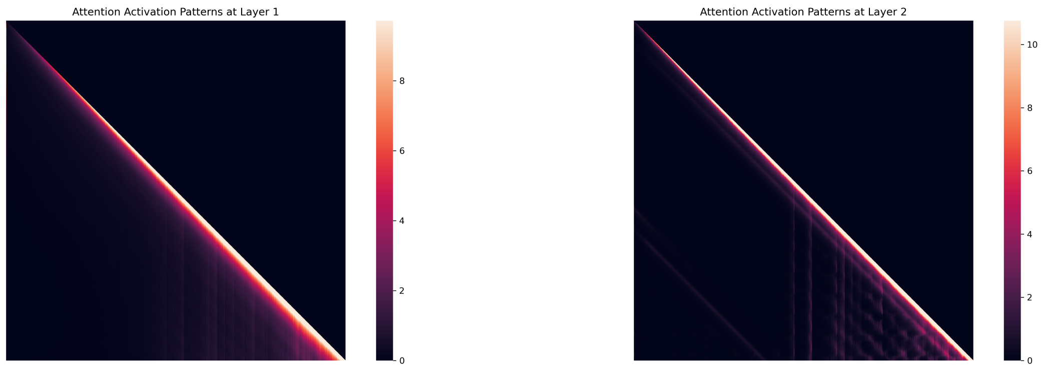


Figure 7.14 Attention activation patterns of the two layers of the Transformer on the in-distribution validation set. For the sake of readability, the attention activation patterns are shown for the first 1024 positions and are compensated by multiplying each row with the number of unmasked positions.

The Longformer performed significantly worse than the LSTM and the Transformer on the CPU out-of-distribution behavior. The attention patterns learned by the Transformer on this behavior depicted in Figure 7.15 reveal that the Transformer learns dependencies that span further than the window size of the Longformer. In other words, this lower-complexity network is unable to model the dependencies learnt by the Transformer. Nonetheless, most of the dependencies learned by the Transformer span less than 400 tokens, which is still in the range of what the LSTM can model [48, 49].

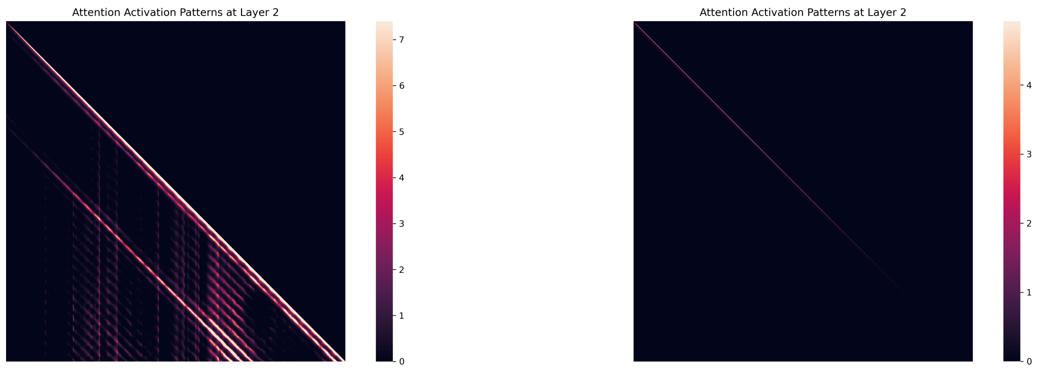


Figure 7.15 Attention activation patterns of the second layer of the Transformer (left) and Longformer (right) on the CPU out-of-distribution validation set. The Transformer leverages positions that the Longformer masks. For the sake of readability, the attention activation patterns are compensated by multiplying each row with the number of unmasked positions.

In order to assess whether the proposed methodology is able to detect small latencies, small delays from 1 microsecond to 1 millisecond were introduced independently at random positions of an in-distribution sample from the validation set. Let us briefly describe the experiment. First, a sample comprising N system calls is drawn from the in-distribution validation set. The perplexity of the original sample (without delay) is evaluated as a baseline. Then, the sample is duplicated $d \times p$ times, where d is the number of delays and p is the number of positions considered. Then, each of the d delays are added to the duration of the system calls corresponding to each of the p positions, thereby allowing to compute of the average perplexity for a given delay. Figure 7.16 depicts the perplexity averaged across positions as a function of the delay. Additional figures for other in-distribution samples and language models are available on GitHub. The average perplexity is always above the baseline depicted by the red horizontal line and increases with the delay. As a result, the proposed methodology is indeed able to detect small latencies, and the detection becomes more effective as delays increase. The lower impact of delays between 10^5 and 10^6 nanoseconds on the perplexity may be caused by the encoding and would necessitate further investigation depending on the use case.

7.5.3 Efficiency

The proposed novelty detection methodology must be efficient to be widely adopted by the community. Table 7.4 reports metrics on the efficiency of the neural networks. The LSTM

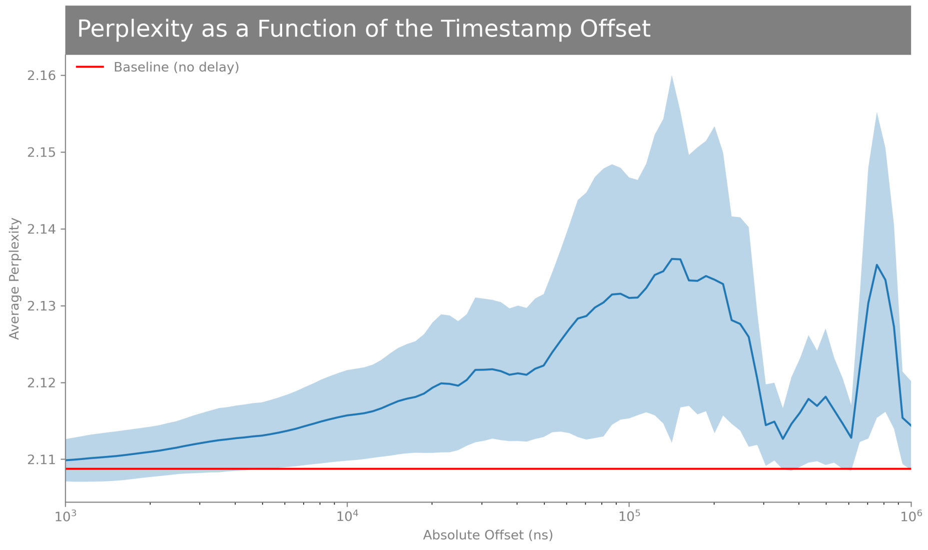


Figure 7.16 The average perplexity of an in-distribution request from the validation set as 100 delays from 1 microsecond to 1 millisecond are introduced at 100 random positions. The shade indicates the standard deviation of the perplexity. The red horizontal line indicates the perplexity of the request without delays.

is the lightest method, only requiring 0.6 Gb of memory, and the Longformer is the fastest method, converging in about 3 hours and processing a batch of 16 samples in only 16 ms at inference time. As expected, the Transformer is the slowest model per batch and has the largest memory footprint due to its quadratic complexity. Nonetheless, the inference for a batch of 16 sequences takes less than 100ms on a single GPU, regardless of the neural network. Consequently, we can claim that the methods are suited to detect novel behaviors in a timely fashion.

Table 7.4 Efficiency Analysis of the Neural Networks.

	Convergence Time	Batch Training (4 GPUs)	Batch Inference (1 GPU)	Memory
LSTM	23h19 ± 1h22	209 ± 4 ms	43 ± 0 ms	0.6 Gb
Transformer	15h51 ± 3h10	397 ± 8 ms	89 ± 1 ms	5.6 Gb
Longformer	3h08 ± 0h07	57 ± 3 ms	16 ± 0 ms	1.8 Gb

7.6 Threats to Validity

This section acknowledges the threats to the validity of the proposed datasets and methodology.

7.6.1 Threats to Internal Validity

Threats to internal validity relate to the soundness of the evaluation methodology and the ability to draw conclusions from the results.

The first two threats to internal validity arise from evaluating the novelty detection methodology on datasets collected for that purpose. First, the in-distribution and out-of-distribution sets may comprise unforeseen differences that facilitated the detection of novel behaviors, thus overestimating the methodology's performance. Second, the in-distribution validation and test sets may contain behaviors that were not in the training set, thus underestimating the methodology's performance. For instance, the in-distribution validation set would comprise a novel behavior if an unexpected network issue occurred at collection time. These threats are due to the cost of manually investigating each dataset. As a mitigation, the datasets were collected in a carefully controlled environment with dedicated machines and compared with simple metrics on the requests, as shown in Table 7.1. Furthermore, the scripts and datasets have been publicly released for researchers and practitioners to investigate.

The third threat to internal validity arises from the balanced nature of the validation and test sets. As explained in Section 7.4.2, novelties are rare in practice. Consequently, the number of positive samples in practice is expected to be much smaller than that of negative samples, which may affect the evaluation. Since the ratio of novelties greatly depends on the use case, we leave it to researchers to sample positive instances based on their use case, and this research considers that novelties occur as frequently as known behavior.

The final threat to internal validity arises from manually tuning the neural networks instead of conducting a grid or random search [195]. The primary reason behind this decision is to reduce the computational cost of the experiments, thereby making them more easily reproducible and reducing the environmental impact of this research. However, the hyperparameters may be suboptimal, and a carefully tuned Transformer may consistently outperform the LSTM. Despite the limited manually tuning, all the evaluated neural networks performed exceptionally well. As a mitigation, the code has been made publicly available and easily reproducible for researchers and practitioners to investigate alternative architectures and hyperparameters.

7.6.2 Threats to External Validity

Threats to external validity relate to the ability to generalize the proposed approach to other use cases. The datasets may not illustrate real world use cases and may not be diverse enough, thus overestimating the performance of the methodology on other real cases. This threat is due to the lack of publicly available modern and massive datasets of kernel traces. In [16], the authors reported that the LSTM and the Transformer achieved comparable language model accuracy on a dataset similar to ours and on a real-world dataset collected by Ciena, indicating that the collected dataset is representative of some real-world use cases. Unfortunately, these two datasets contain a single behavior and are thus not suited to evaluate our novelty detection methodology. As a mitigation, the use cases were designed to be realistic and of genuine interest. Additionally, the entire project and the trained models have been made public for researchers and practitioners to evaluate on their private datasets.

7.7 Discussion

This section discusses the strengths and acknowledges the limitations of the proposed methodologies that may hinder their adoption.

7.7.1 Strengths and Benefits

This section briefly discusses the three main benefits of the proposed methodology.

First, the proposed approach is data agnostic since a language model is a probability distribution over sequences of tokens. This work focused on system calls as they expose the system behavior and do not require manually instrumenting the applications considered. Nonetheless, tokens need not be system calls. One may learn a language model of userspace events, scheduler events, log lines, or whatever is suited for the use case at hand.

Second, the approach is novelty agnostic since the detection relies on the perplexity of the sequence under the model. Any deviation from previously observed behavior is likely to increase the perplexity and is thus detectable, including component upgrades, software updates, new users, rare queries, misconfigurations, latency, intrusions, hardware failures, and bugs.

Finally, the approach does not heavily depend on an expert after the data collection to annotate the data with labels that are error-prone or to extract high-level features from the trace that are often suboptimal. Although the neural networks have hyperparameters that must be tuned beforehand to achieve the best performance, multiple techniques such as random search [195] have been proposed to find them automatically.

7.7.2 Limitations and Shortcomings

The primary limitations and shortcomings of the proposed methodology are the limited interpretability, the risk of leaking information, the sensitivity to repeated strings attacks, and the potentially high environmental impact.

Neural networks are often considered black boxes because their decisions are hard to explain. Interpretability is an active research avenue: for instance, researchers still debate whether attention weights are interpretable [157, 158, 156]. Although the proposed approach is unable to justify the detection, the predicted conditional probability of the individual system calls may indicate the location of the root cause of the novelty.

As explained by Carlini et al. [198], large language models may leak exact training samples. Indeed, by asking the model to generate sequences, exact samples from the training set may be generated since their likelihood has been maximized during training. However, this potential privacy issue is not severe, in our opinion, as one would need access to the model to generate samples and there is no indication which of the generated samples are actually part of the training data. Furthermore, the networks are only able to generate sequences of system call names without their arguments.

The Transformer is known to be sensitive to false negatives that are samples containing repeated strings [199], which means that a carefully designed attack that would repeat many times a sequence of events may not be detected. However, these types of attacks could be easily detected with simple metrics such as the number of system calls or the duration of the request.

Finally, neural networks are costly and emit a significant amount of carbon dioxide (CO2). Strubell et al. [79] estimated that training a Transformer with neural architecture search generates up to 284,000 kg of CO2. For reference, the average American emits 16,400 kg of CO2 annually, and the average car emits about 57,200 kg during its lifetime (fuel included).

7.7.3 Future Work

This section discusses some of the most exciting and promising research directions that would be necessary to deploy the proposed methodology.

The behavior of computer systems frequently evolves. In other words, the set of known behaviors is continuously expanding, and consequently, the language models should be continuously updated. Tsimpoukelli et al. [200] observed that "when trained at sufficient scale, autoregressive language models exhibit the notable ability to learn a new language task after being prompted with just a few examples." Accordingly, the first research avenue is to scale the language models and train them on significantly more behaviors to assess whether they are able to assimilate new behaviors with just a few samples.

Novel behavior should be detected as they occur. However, scaling neural networks typically increases the amount of computation and memory required. Consequently, the second research avenue is to scale the neural networks without increasing the amount of computation with a mixture of experts [109]. The idea is to train multiple networks called experts and a router that forwards the input to a fixed number of relevant experts. As a result, increasing the number of experts increases the model size while keeping the computational cost constant.

Finally, the models should be interpretable and robust. Attention weights may not explain the output, at least not in a straightforward manner, but there have been attempts at improving their interpretability, notably by averaging attention scores [201]. Robustness has always been a concern, and multiple techniques have been proposed. One of the most interesting, in our opinion, is sharpness-aware minimization (SAM) [202] which seeks parameters in neighborhoods with uniformly low loss.

7.8 Conclusion

This paper introduces a data and novelty-agnostic language model-based approach to detecting novelties from system call sequences. The proposed methodology was evaluated using three neural networks: the widely popular LSTM, the state-of-the-art Transformer, and the lower-complexity Longformer. The three models were able to detect six novel behaviors effectively. Interestingly, the inductive bias of the LSTM toward local dependencies helped the model achieve better novelty detection performance on 3 out of 6 behaviors when compared to the more flexible Transformer and Longformer. Crucially, this observation applies solely to our dataset and may arise from the short nature of the requests collected.

Finally, this paper also introduces a new open-source dataset of kernel traces comprising over 2 million web requests with seven distinct behaviors, as large neural networks are known to benefit greatly from larger datasets. Due to the lack of publicly available equivalent datasets, the data collection scripts, the trained models, and the source code have been publicly released for researchers and practitioners to evaluate the proposed approach on their use cases.

7.9 Acknowledgment

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena, AMD, and EfficiOS for funding this project.

CHAPTER 8 GENERAL DISCUSSION

First, this chapter summarizes the three contributions of this thesis that aim at detecting anomalies and novelties in computer systems from kernel traces. Then, this chapter acknowledges and discusses the shortcomings of the three contributions and, more broadly, of this thesis.

8.1 Summary of Works

In Chapter 5, the execution states of the threads that contribute to the response time of requests are first extracted from the critical path. Then, based on these hand-crafted higher-level features, abnormal requests are identified with DBSCAN and subsequently grouped with k-means. Finally, each cluster of outliers is investigated separately with simple yet effective statistics on the requests, such as the number of distinct system calls and n-grams. The limited scope of the first contribution of this thesis allowed getting familiar with the data and task at hand while identifying latency issues and revealing their potential root cause. Most notably, the proposed approach correctly identified a genuine PHP contention issue.

In Chapter 6, a joint representation of the system call names along with their arguments is learned using both embedding and encoding. The additional information allowed the neural networks to make more informed and, ultimately, more precise predictions. Indeed, the language modeling performance of two widely popular neural networks is significantly improved by leveraging the additional information that is readily available. The ablation study revealed that the improvement could not be explained solely by the increase in embedding size, indicating that the networks were indeed able to leverage the arguments. Notably, the cross-entropy of the Transformer on the masked language model, a popular pre-training objective, decreased from 0.485 to 0.182 (-62.5%), while the accuracy increased from 82.8% to 94.1% (+13.6%). As expected, arguments relating to the process that issued the call produced the highest overall impact on the performance. Most importantly, the increase in size yielded a reasonable impact on the inference speed. For instance, the epoch time of the Transformer during optimization for the masked language model increased from 232.2ms to 238.5ms[1] (+2.7%).

In Chapter 7, the representation introduced by the second contribution is employed to detect novelties in kernel traces with a language model of the system calls. The proposed approach

[1]Please note that the code has not been heavily optimized.

benefits from minimal expert hand-crafting while being data- and novelty-agnostic. Three neural networks have been investigated with the left-to-right language model: the LSTM, the Transformer, and the Longformer. Each model effectively detected the six novelties collected with an area under the ROC curve and an F-score greater than 95% in most experiments. Furthermore, the models were able to detect small latencies ranging from 1 microsecond to 1 millisecond randomly inserted into the requests, indicating that our approach is suitable for detecting performance issues. Contrary to our expectations, the LSTM performed similarly to the Transformer, although the latter has the ability to model arbitrary-length dependencies. A visualization of the activation patterns of the Transformer's attention revealed that the dependencies learned were primarily local, indicating that the dataset contains few extremely long dependencies. Nonetheless, researchers and practitioners are encouraged to assess the type of dependencies in their data before discarding the more flexible networks.

8.2 Limitations

The first and foremost limitation of this research is that most experiments were conducted on datasets collected by ourselves. This decision comes from the lack of publicly available large and modern datasets of kernel traces. Indeed, the usual datasets such as UNM [182], KDD98 [183], and ADFA-LD [184] are small and obsolete as explained by Creech and Hu [184] and Murtaza et al. [13]. Furthermore, they omit the arguments of the system calls, which have been at the core of the second and third contributions of this thesis. Collecting the dataset ourselves poses a threat to this research's internal and external validity. Indeed, the traces may not represent real-world use cases or contain unexpected behaviors that would impact the evaluation. As a mitigation, the partnering company was frequently involved in this research to attest to the soundness of the collected datasets and proposed methodologies. Furthermore, the approaches were evaluated whenever possible on datasets collected by the partnering company on in-production servers.

Due to the exploratory nature of this research, the proposed methodologies would necessitate additional development before their real-world deployment. Discussions with the partnering company revealed three critical aspects of the proposed approaches that must be improved in order for them to be adopted: efficiency, robustness, and explainability. Indeed, each neural network has been trained on multiple high-end GPUs for up to 3 days, making the approaches resource-intensive and expensive to deploy in practice. Moreover, the neural networks are sensitive to their hyperparameters, to noise, and to attacks such as adversarial samples or repeated strings in the case of the Transformer [199]. Nonetheless, we believe that these potential issues are not severe as the former requires the attacker to access the model's parameters, while the latter could be detected with simple metrics such as the number of system calls. Finally, neural networks are often described as black-box due to the difficulty of explaining their decision. The lack of transparency has been one of the most common complaints from practitioners who typically require strong guarantees.

As briefly mentioned, neural networks require significant computational resources, which come at a high price but also have a high impact on the environment. For instance, Strubell et al. [79] estimated that training a single large Transformer called BERT [51] on GPU produces about the same amount of CO2 as a trans-American flight. Due to the exploratory nature of this research, constraints on the computational resources, and environmental considerations, no extensive grid search or random search was conducted. As a consequence, the models may be suboptimal, and the performance reported may be significantly lower than what could be achieved with more resources.

CHAPTER 9 CONCLUSION AND RECOMMENDATIONS

In response to the aforementioned limitations, this section concludes this thesis by discussing the promising research avenues to extend and improve this research.

Let us first discuss interesting research avenues to address the limitations raised by the partnering company: the robustness, efficiency, and explainability of the neural networks. One of the most straightforward techniques to improve the robustness of machine learning models is *ensembling*, also referred to as ensemble methods. The core idea behind ensembling is that the aggregation of multiple machine learning models achieves a better performance than any individual model. In the case of this research, multiple neural networks could be randomly initiated with distinct seeds and potentially trained on a subset of the samples and features. Then, the average of the perplexities estimated by each model should be more robust to noise. Notably, ensembling has already been used with great success to improve the robustness of an LSTM-based intrusion detection method from kernel traces [2]. Another promising technique for neural networks is *sharpness-aware minimization* (SAM) [202], which seeks parameters that lie in neighborhoods having uniformly low loss, thereby making the model less sensitive to noise. Numerous techniques have been proposed to improve the efficiency of neural networks, as extensively investigated in Chapter 3. In our opinion, researchers should first explore straightforward techniques compatible with ensembling, such as a good initialization strategy and knowledge distillation. The latter technique transfers the knowledge from a large model or an ensemble of models called teacher(s) to a single, typically smaller, model called the student by training the student to reproduce the output of the teacher. Furthermore, quantization may be an interesting research avenue depending on the hardware available. Finally, the explainability of neural networks remains an open problem as of the writing of this thesis. One of the most active research directions is whether attention weights are interpretable [157, 158, 156]. In our opinion, attention weights and activation patterns are helpful as insight rather than explanation. Nonetheless, they should be taken with caution and further investigated.

Finally, computer systems and their behaviors continuously evolve. In other words, the set of known behaviors is continuously expanding. Accordingly, machine learning models should be continuously updated to take into account the latest behaviors. The fields of online learning, continual learning, and open-set learning have been particularly active in recent years, and numerous approaches have been proposed. Interestingly, one potential solution came from scaling the neural networks. Evidently, increasing the number of parameters has been shown to improve the performance of neural networks, especially for natural language processing [50, 193]. Furthermore, sufficiently large language models have exhibited the interesting ability to learn a new language after being prompted with just a few examples. Therefore, scaling the system call language model at the core of the second and third contributions could provide a simple approach to adapting to new behaviors with limited resources.

REFERENCES

[1] D. Spinellis, “Trace: A tool for logging operating system call transactions,” *SIGOPS Oper. Syst. Rev.*, vol. 28, no. 4, p. 56–63, oct 1994. [Online]. Available: https://doi.org/10.1145/191525.191540

[2] G. Kim *et al.*, “Lstm-based system-call language modeling and robust ensemble method for designing host-based intrusion detection systems,” 11 2016. [Online]. Available: https://arxiv.org/abs/1611.01726

[3] M. Du *et al.*, “Deeplog: Anomaly detection and diagnosis from system logs through deep learning,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1285–1298. [Online]. Available: https://doi.org/10.1145/3133956.3134015

[4] A. Brown *et al.*, “Recurrent neural network attention mechanisms for interpretable system log anomaly detection,” *CoRR*, vol. abs/1803.04967, 2018.

[5] J. Bogatinovski *et al.*, “Self-supervised anomaly detection from distributed traces,” in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, 2020, pp. 342–347.

[6] H. Guo, S. Yuan, and X. Wu, “Logbert: Log anomaly detection via bert,” 2021. [Online]. Available: https://arxiv.org/abs/2103.04475

[7] K. Araki, Z. Furukawa, and J. Cheng, “A general framework for debugging,” *IEEE Software*, vol. 8, no. 3, pp. 14–20, 1991.

[8] J. M. Anderson *et al.*, “Continuous profiling: Where have all the cycles gone?” *ACM Trans. Comput. Syst.*, vol. 15, no. 4, p. 357–390, nov 1997. [Online]. Available: https://doi.org/10.1145/265924.265925

[9] M. Gebai and M. R. Dagenais, “Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead,” *ACM Comput. Surv.*, vol. 51, no. 2, mar 2018.

[10] M. Desnoyers and M. R. Dagenais, “The lttng tracer: A low impact performance and behavior monitor for gnu/linux,” in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

[11] M. Desnoyers, "Low-impact operating system tracing," Ph.D. dissertation, École Polytechnique de Montréal, 2009.

[12] C. Harper-Cyr, M. R. Dagenais, and A. S. Bushehri, "Fast and flexible tracepoints in x86," *Software: Practice and Experience*, vol. 49, no. 12, pp. 1712–1727, 2019.

[13] S. S. Murtaza *et al.*, "A host-based anomaly detection approach by representing system calls as states of kernel modules," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 431–440.

[14] G. Tandon and P. K. Chan, "Learning useful system call attributes for anomaly detection," in *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference, Clearwater Beach, Florida, {USA}*, 2005, pp. 405–411.

[15] ——, "On the learning of system call attributes for host-based anomaly detection," *International Journal on Artificial Intelligence Tools*, vol. 15, no. 06, pp. 875–892, 2006.

[16] Q. Fournier *et al.*, "On improving deep learning trace analysis with system call arguments," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 120–130.

[17] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection from system tracing data using multimodal deep learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, July 2019, pp. 179–186.

[18] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection and classification using distributed tracing and deep learning," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 241–250.

[19] O. M. Ezeme, Q. Mahmoud, and A. Azim, "A framework for anomaly detection in time-driven and event-driven processes using kernel traces," *IEEE Transactions on Knowledge and Data Engineering*, p. 1, 2020.

[20] M. Dymshits, B. Myara, and D. Tolpin, "Process monitoring on sequences of system call count vectors," *CoRR*, vol. abs/1707.0, 2017.

[21] A. Liu *et al.*, "A comparison of system call feature representations for insider threat detection," in *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, jun 2005, pp. 340–347.

[22] Dae-Ki Kang, D. Fuller, and V. Honavar, "Learning classifiers for misuse and anomaly detection using a bag of system calls representation," in *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, jun 2005, pp. 118–125.

[23] R. Canzanese, S. Mancoridis, and M. Kam, "System call-based detection of malicious processes," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, Aug 2015, pp. 119–124.

[24] Z. Xu *et al.*, "A multi-module anomaly detection scheme based on system call prediction," in *2013 IEEE 8th Conference on Industrial Electronics and Applications (ICIEA)*, 2013, pp. 1376–1381.

[25] C. Wressnegger *et al.*, "A close look on n-grams in intrusion detection: Anomaly detection vs. classification," in *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, ser. AISec '13. New York, NY, USA: ACM, 2013, pp. 67–76. [Online]. Available: http://doi.acm.org/10.1145/2517312.2517316

[26] S. Lv *et al.*, "Intrusion prediction with system-call sequence-to-sequence model," *CoRR*, vol. abs/1808.01717, 2018.

[27] Q. Fournier *et al.*, "Automatic cause detection of performance problems in web applications," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019, pp. 398–405.

[28] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, http://www.deeplearningbook.org.

[29] A. M. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, no. 236, pp. 433–460, 1950. [Online]. Available: http://www.jstor.org/stable/2251299

[30] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.

[31] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[32] R. Dechter, "Learning while searching in constraint-satisfaction-problems," in *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, ser. AAAI'86. AAAI Press, 1986, p. 178–183.

[33] A. G. Ivakhnenko and V. G. Lapa, *Cybernetic Predicting Devices.* CCM Information Corporation, 1965.

[34] K. Asmitha and P. Vinod, "A machine learning approach for linux malware detection," *Proceedings of the 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques, ICICT 2014*, pp. 825–830, 2014.

[35] S. S. Murtaza *et al.*, "On the comparison of user space and kernel space traces in identification of software anomalies," in *2012 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 127–136.

[36] F. Song *et al.*, "Exad: A system for explainable anomaly detection on big data traces," in *ICDMW 2018 - IEEE International Conference on Data Mining Workshops*, Singapore, Singapore, Nov. 2018.

[37] D. E. Rumelhart *et al.*, *Schemata and Sequential Thought Processes in PDP Models.* Cambridge, MA, USA: MIT Press, 1986, p. 7–57.

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

[39] K. Cho *et al.*, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *EMNLP.* Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734. [Online]. Available: https://www.aclweb.org/anthology/D14-1179

[40] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani *et al.*, Eds. Curran Associates, Inc., 2014, pp. 3104–3112.

[41] J. Cheng, L. Dong, and M. Lapata, "Long short-term memory-networks for machine reading," in *EMNLP.* Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 551–561. [Online]. Available: https://www.aclweb.org/anthology/D16-1053

[42] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014.

[43] A. Vaswani *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems 30*, I. Guyon *et al.*, Eds. Curran Associates, Inc., 2017, pp. 5998–6008. [Online]. Available: http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

[44] C. C. Aggarwal, *Outlier Analysis*, 2nd ed. Springer Publishing Company, Incorporated, 2016.

[45] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *ICLR*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.0473

[46] A. Galassi, M. Lippi, and P. Torroni, "Attention in natural language processing," *IEEE Transactions on Neural Networks and Learning Systems*, p. 1–18, 2020. [Online]. Available: http://dx.doi.org/10.1109/TNNLS.2020.3019893

[47] L. Weng, "Attention? attention!" *lilianweng.github.io/lil-log*, 2018. [Online]. Available: http://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

[48] Z. Dai *et al.*, "Transformer-XL: Attentive language models beyond a fixed-length context," in *ACL*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 2978–2988. [Online]. Available: https://www.aclweb.org/anthology/P19-1285

[49] U. Khandelwal *et al.*, "Sharp nearby, fuzzy far away: How neural language models use context," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 284–294. [Online]. Available: https://www.aclweb.org/anthology/P18-1027

[50] T. Brown *et al.*, "Language models are few-shot learners," in *NeurIPS*, H. Larochelle *et al.*, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[51] J. Devlin *et al.*, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://www.aclweb.org/anthology/N19-1423

[52] N. Carion *et al.*, "End-to-end object detection with transformers," in *ECCV*, A. Vedaldi *et al.*, Eds. Cham: Springer International Publishing, 2020, pp. 213–229.

[53] A. Dosovitskiy *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," in *ICLR*, 2021. [Online]. Available: https://openreview.net/forum?id=YicbFdNTTy

[54] S. Khan *et al.*, "Transformers in vision: A survey," *ACM Comput. Surv.*, dec 2021, just Accepted. [Online]. Available: https://doi.org/10.1145/3505244

[55] A. Gulati *et al.*, “Conformer: Convolution-augmented transformer for speech recognition,” in *Proc. Interspeech 2020*, 2020, pp. 5036–5040.

[56] C. Subakan *et al.*, “Attention is all you need in speech separation,” in *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021, pp. 21–25.

[57] Q. Zhang *et al.*, “Transformer transducer: A streamable speech recognition model with transformer encoders and rnn-t loss,” in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020, pp. 7829–7833.

[58] M. Zaheer *et al.*, “Big bird: Transformers for longer sequences,” in *Advances in Neural Information Processing Systems*, H. Larochelle *et al.*, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 17 283–17 297. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/c8512d142a2d849725f31a9a7a361ab9-Paper.pdf

[59] S. Karita *et al.*, “A comparative study on transformer vs rnn in speech applications,” in *ASRU*, 2019, pp. 449–456.

[60] A. Radford and K. Narasimhan, “Improving language understanding by generative pre-training,” 2018.

[61] C. Liu *et al.*, “Improving RNN transducer based ASR with auxiliary tasks,” in *IEEE Spoken Language Technology Workshop, SLT 2021, Shenzhen, China, January 19-22, 2021.* IEEE, 2021, pp. 172–179. [Online]. Available: https://doi.org/10.1109/SLT48900.2021.9383548

[62] N. Shazeer *et al.*, “Outrageously large neural networks: The sparsely-gated mixture-of-experts layer.” in *ICLR (Poster).* OpenReview.net, 2017. [Online]. Available: http://dblp.uni-trier.de/db/conf/iclr/iclr2017.html#ShazeerMMDLHD17

[63] Q. Xie *et al.*, “Self-training with noisy student improves imagenet classification.” in *CVPR.* IEEE, 2020, pp. 10 684–10 695. [Online]. Available: http://dblp.uni-trier.de/db/conf/cvpr/cvpr2020.html#XieLHL20

[64] A. Kolesnikov *et al.*, “Big transfer (bit): General visual representation learning,” in *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part V*, ser. Lecture Notes in Computer Science, A. Vedaldi *et al.*, Eds., vol. 12350. Springer, 2020, pp. 491–507. [Online]. Available: https://doi.org/10.1007/978-3-030-58558-7%5F29

[65] S. Gray, A. Radford, and D. P. Kingma, “Gpu kernels for block-sparse weights,” 2017.

[66] A. Wang *et al.*, “GLUE: A multi-task benchmark and analysis platform for natural language understanding,” in *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 353–355. [Online]. Available: https://www.aclweb.org/anthology/W18-5446

[67] W. Chan *et al.*, “Listen, attend and spell: A neural network for large vocabulary conversational speech recognition,” in *ICASSP*, 2016, pp. 4960–4964.

[68] D. S. Park *et al.*, “Specaugment on large scale datasets,” in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020, pp. 6879–6883.

[69] Y. Liu *et al.*, “Roberta: A robustly optimized BERT pretraining approach,” *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[70] Y. Tay *et al.*, “Efficient transformers: A survey,” *CoRR*, vol. abs/2009.06732, 2020. [Online]. Available: https://arxiv.org/abs/2009.06732

[71] T. Lin *et al.*, “A survey of transformers,” *CoRR*, vol. abs/2106.04554, 2021. [Online]. Available: https://arxiv.org/abs/2106.04554

[72] I. O. Tolstikhin *et al.*, “Mlp-mixer: An all-mlp architecture for vision,” *CoRR*, vol. abs/2105.01601, 2021. [Online]. Available: https://arxiv.org/abs/2105.01601

[73] H. Liu *et al.*, “Pay attention to mlps,” *ArXiv*, vol. abs/2105.08050, 2021.

[74] W. Yu *et al.*, “Metaformer is actually what you need for vision,” in *CVPR*, June 2022, pp. 10 819–10 829.

[75] K. He *et al.*, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[76] J. Lei Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” *arXiv e-prints*, p. arXiv:1607.06450, 2016.

[77] Z. Li, K. Lyu, and S. Arora, “Reconciling modern deep learning with traditional optimization analyses: The intrinsic learning rate,” in *Advances in Neural Information Processing Systems*, H. Larochelle *et al.*, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 14 544–14 555. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/a7453a5f026fb6831d68bdc9cb0edcae-Paper.pdf

[78] S. Santurkar *et al.*, “How does batch normalization help optimization?” in *Advances in Neural Information Processing Systems*, S. Bengio *et al.*, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/905056c1ac1dad141560467e0a99e1cf-Paper.pdf

[79] E. Strubell, A. Ganesh, and A. McCallum, “Energy and policy considerations for modern deep learning research,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 09, pp. 13 693–13 696, Apr. 2020. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/7123

[80] IEA, “World gross electricity production, by source, 2018,” 2018. [Online]. Available: https://www.iea.org/data-and-statistics/charts/world-gross-electricity-production-by-source-2018

[81] Y. LeCun, J. S. Denker, and S. A. Solla, “Optimal brain damage,” in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. Morgan-Kaufmann, 1990, pp. 598–605. [Online]. Available: http://papers.nips.cc/paper/250-optimal-brain-damage.pdf

[82] T. Chen *et al.*, “Training deep nets with sublinear memory cost,” *CoRR*, vol. abs/1604.06174, 2016. [Online]. Available: http://arxiv.org/abs/1604.06174

[83] OpenAI, “Saving memory using gradient-checkpointing,” https://github.com/openai/gradient-checkpointing, 2013.

[84] L. Dinh, D. Krueger, and Y. Bengio, “NICE: non-linear independent components estimation,” in *ICLR*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1410.8516

[85] A. N. Gomez *et al.*, “The reversible residual network: Backpropagation without storing activations,” in *Advances in Neural Information Processing Systems*, I. Guyon *et al.*, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/f9be311e65d81a9ad8150a60844bb94c-Paper.pdf

[86] L. Dinh, J. Sohl-Dickstein, and S. Bengio, “Density estimation using real NVP,” in *ICLR*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=HkpbnH9lx

[87] N. Kitaev, L. Kaiser, and A. Levskaya, “Reformer: The efficient transformer,” in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=rkgNKkHtvB

[88] S. Wang *et al.*, “Linformer: Self-attention with linear complexity,” *arXiv e-prints*, p. arXiv:2006.04768, Jun. 2020.

[89] Z. Lan *et al.*, “Albert: A lite bert for self-supervised learning of language representations,” in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=H1eA7AEtvS

[90] H. Sajjad *et al.*, “On the effect of dropping layers of pre-trained transformer models,” *arXiv e-prints*, p. arXiv:2004.03844, Apr. 2020.

[91] P. Michel, O. Levy, and G. Neubig, “Are sixteen heads really better than one?” in *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/2c601ad9d2ff9bc8b282670cdd54f69f-Paper.pdf

[92] J. Frankle and M. Carbin, “The lottery ticket hypothesis: Finding sparse, trainable neural networks,” in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=rJl-b3RcF7

[93] S. Prasanna, A. Rogers, and A. Rumshisky, “When BERT Plays the Lottery, All Tickets Are Winning,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Online: Association for Computational Linguistics, Nov. 2020, pp. 3208–3229. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-main.259

[94] J. Ba and R. Caruana, “Do deep nets really need to be deep?” in *NIPS*, Z. Ghahramani *et al.*, Eds., vol. 27. Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/ea8fcd92d59581717e06eb187f10666d-Paper.pdf

[95] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” *arXiv e-prints*, p. arXiv:1503.02531, Mar. 2015.

[96] V. Sanh *et al.*, “Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter,” *CoRR*, vol. abs/1910.01108, 2019. [Online]. Available: http://arxiv.org/abs/1910.01108

[97] H. Tsai *et al.*, “Small and practical BERT models for sequence labeling,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and*

the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP). Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 3632–3636. [Online]. Available: https://www.aclweb.org/anthology/D19-1374

[98] X. Jiao *et al.*, "TinyBERT: Distilling BERT for natural language understanding," in *Findings of the Association for Computational Linguistics: EMNLP 2020.* Online: Association for Computational Linguistics, Nov. 2020, pp. 4163–4174. [Online]. Available: https://www.aclweb.org/anthology/2020.findings-emnlp.372

[99] A. Warstadt, A. Singh, and S. R. Bowman, "Neural network acceptability judgments," *Transactions of the Association for Computational Linguistics*, vol. 7, pp. 625–641, Mar. 2019. [Online]. Available: https://www.aclweb.org/anthology/Q19-1040

[100] P. Micikevicius *et al.*, "Mixed precision training," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=r1gs9JgRZ

[101] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[102] Y. Bengio, "Estimating or propagating gradients through stochastic neurons," *CoRR*, vol. abs/1305.2982, 2013. [Online]. Available: http://arxiv.org/abs/1305.2982

[103] O. Zafrir *et al.*, "Q8bert: Quantized 8bit bert," 2019.

[104] P. Rajpurkar *et al.*, "SQuAD: 100,000+ questions for machine comprehension of text," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing.* Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2383–2392. [Online]. Available: https://www.aclweb.org/anthology/D16-1264

[105] P. Stock *et al.*, "Training with quantization noise for extreme model compression," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=dV19Yyi1fS3

[106] S. Merity *et al.*, "Pointer sentinel mixture models," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.* OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=Byj72udxe

[107] Y. Huang *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf

[108] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, 07–09 Jul 2015, pp. 448–456. [Online]. Available: http://proceedings.mlr.press/v37/ioffe15.html

[109] R. A. Jacobs *et al.*, "Adaptive mixtures of local experts," *Neural Computation*, vol. 3, pp. 79–87, 1991.

[110] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *arXiv e-prints*, p. arXiv:2101.03961, Jan. 2021.

[111] K. Clark *et al.*, "Electra: Pre-training text encoders as discriminators rather than generators," in *ICLR*, 2020. [Online]. Available: https://openreview.net/forum?id=r1xMH1BtvB

[112] T. Wolf *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations.* Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[113] W. L. Taylor, "“cloze procedure”: A new tool for measuring readability," *Journalism Quarterly*, vol. 30, no. 4, pp. 415–433, 1953. [Online]. Available: https://doi.org/10.1177/107769905303000401

[114] L. Liu *et al.*, "Understanding the difficulty of training transformers," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, B. Webber *et al.*, Eds. Association for Computational Linguistics, 2020, pp. 5747–5763. [Online]. Available: https://doi.org/10.18653/v1/2020.emnlp-main.463

[115] X. S. Huang *et al.*, "Improving transformer optimization through better initialization," in *Proceedings of the 37th International Conference on Machine Learning,*

ICML 2020, 13-18 July 2020, Virtual Event, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 4475–4483. [Online]. Available: http://proceedings.mlr.press/v119/huang20f.html

[116] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html

[117] L. Liu *et al.*, "On the variance of the adaptive learning rate and beyond," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=rkgz2aEKDr

[118] R. Xiong *et al.*, "On layer normalization in the transformer architecture," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 10 524–10 533. [Online]. Available: https://proceedings.mlr.press/v119/xiong20b.html

[119] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=r1Ue8Hcxg

[120] E. Real *et al.*, "Regularized evolution for image classifier architecture search," in *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 4780–4789. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.33014780

[121] H. Liu, K. Simonyan, and Y. Yang, "DARTS: differentiable architecture search," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=S1eYHoC5FX

[122] B. Zoph *et al.*, "Learning transferable architectures for scalable image recognition," in *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June*

18-22, 2018. IEEE Computer Society, 2018, pp. 8697–8710. [Online]. Available: http://openaccess.thecvf.com/content%5Fcvpr%5F2018/html/Zoph%5FLearning%5FTransferable%5FArchitectures%5FCVPR%5F2018%5Fpaper.html

[123] H. Pham *et al.*, "Efficient neural architecture search via parameter sharing," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. Proceedings of Machine Learning Research, J. G. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 4092–4101. [Online]. Available: http://proceedings.mlr.press/v80/pham18a.html

[124] N. Srivastava *et al.*, "Dropout: a simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2670313

[125] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *JMLR*, vol. 20, no. 55, pp. 1–21, 2019. [Online]. Available: http://jmlr.org/papers/v20/18-598.html

[126] D. R. So, Q. V. Le, and C. Liang, "The evolved transformer," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 2019, pp. 5877–5886. [Online]. Available: http://proceedings.mlr.press/v97/so19a.html

[127] O. Bojar *et al.*, "Findings of the 2014 workshop on statistical machine translation," in *SIGMT*. Baltimore, Maryland, USA: Association for Computational Linguistics, Jun. 2014, pp. 12–58. [Online]. Available: http://www.aclweb.org/anthology/W/W14/W14-3302

[128] H. Tsai *et al.*, "Finding fast transformers: One-shot neural architecture search by component composition," *arXiv e-prints*, p. arXiv:2008.06808, Aug. 2020.

[129] Z. Wu *et al.*, "Lite transformer with long-short range attention," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=ByeMPlHKPH

[130] Y. Bengio, "Deep learning of representations: Looking forward," in *SLSP*, ser. Lecture Notes in Computer Science, A. Dediu *et al.*, Eds., vol. 7978. Springer, 2013, pp. 1–37. [Online]. Available: https://doi.org/10.1007/978-3-642-39593-2%5F1

[131] A. Graves, “Adaptive computation time for recurrent neural networks,” *CoRR*, vol. abs/1603.08983, 2016. [Online]. Available: http://arxiv.org/abs/1603.08983

[132] M. Dehghani *et al.*, “Universal transformers,” in *ICLR*, 2019. [Online]. Available: https://openreview.net/forum?id=HyzdRiR9Y7

[133] M. Elbayad *et al.*, “Depth-adaptive transformer,” in *ICLR*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=SJg7KhVKPH

[134] S. Sukhbaatar *et al.*, “Adaptive attention span in transformers,” in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 331–335. [Online]. Available: https://www.aclweb.org/anthology/P19-1032

[135] M. Mahoney, “Large text compression benchmark,” 2011. [Online]. Available: http://mattmahoney.net/dc/text.html

[136] M. Li, C. Zorila, and R. Doddipatla, “Transformer-based online speech recognition with decoder-end adaptive computation steps,” *arXiv e-prints*, p. arXiv:2011.13834, Nov. 2020.

[137] Q. Guo *et al.*, “Star-transformer,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 1315–1325. [Online]. Available: https://doi.org/10.18653/v1/n19-1133

[138] R. Child *et al.*, “Generating long sequences with sparse transformers,” *CoRR*, vol. abs/1904.10509, 2019. [Online]. Available: http://arxiv.org/abs/1904.10509

[139] C. Wang *et al.*, “Transformer on a diet,” *arXiv e-prints*, p. arXiv:2002.06170, Feb. 2020.

[140] S. Li *et al.*, “Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting,” in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach *et al.*, Eds., 2019, pp. 5244–5254. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/6775a0635c302542da2c32aa19d86be0-Abstract.html

[141] J. Qiu *et al.*, "Blockwise self-attention for long document understanding," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 2555–2565. [Online]. Available: https://www.aclweb.org/anthology/2020.findings-emnlp.232

[142] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv e-prints*, p. arXiv:2004.05150, Apr. 2020.

[143] Y. Tay *et al.*, "Sparse Sinkhorn attention," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 9438–9447. [Online]. Available: http://proceedings.mlr.press/v119/tay20a.html

[144] H. Shi *et al.*, "Sparsebert: Rethinking the importance analysis in self-attention," in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 2021, pp. 9547–9557. [Online]. Available: http://proceedings.mlr.press/v139/shi21a.html

[145] G. M. Correia, V. Niculae, and A. F. T. Martins, "Adaptively sparse transformers," in *EMNLP-IJCNLP*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 2174–2184. [Online]. Available: https://aclanthology.org/D19-1223

[146] C. J. Maddison, A. Mnih, and Y. W. Teh, "The concrete distribution: A continuous relaxation of discrete random variables," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=S1jE5L5gl

[147] S. Hooker, "The hardware lottery," *CoRR*, vol. abs/2009.06489, 2020. [Online]. Available: https://arxiv.org/abs/2009.06489

[148] A. Roy *et al.*, "Efficient content-based sparse attention with routing transformers," *Trans. Assoc. Comput. Linguistics*, vol. 9, pp. 53–68, 2021. [Online]. Available: https://transacl.org/ojs/index.php/tacl/article/view/2405

[149] Y. Tay *et al.*, "Synthesizer: Rethinking self-attention in transformer models," *arXiv e-prints*, p. arXiv:2005.00743, May 2020.

[150] Y. Xiong *et al.*, "Nyströmformer: A Nyström-Based Algorithm for Approximating Self-Attention," *arXiv e-prints*, p. arXiv:2102.03902, Feb. 2021.

[151] K. M. Choromanski *et al.*, "Rethinking attention with performers," in *ICLR*, 2021. [Online]. Available: https://openreview.net/forum?id=Ua6zuk0WRH

[152] A. Katharopoulos *et al.*, "Transformers are RNNs: Fast autoregressive transformers with linear attention," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 5156–5165. [Online]. Available: http://proceedings.mlr.press/v119/katharopoulos20a.html

[153] A. Vyas, A. Katharopoulos, and F. Fleuret, "Fast transformers with clustered attention," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle *et al.*, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/f6a8dd1c954c8506aadc764cc32b895e-Abstract.html

[154] J. W. Rae *et al.*, "Compressive transformers for long-range sequence modelling," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=SylKikSYDH

[155] Z. Dai *et al.*, "Funnel-transformer: Filtering out sequential redundancy for efficient language processing," in *NeurIPS*, H. Larochelle *et al.*, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/2cd2915e69546904e4e5d4a2ac9e1652-Abstract.html

[156] S. Wiegreffe and Y. Pinter, "Attention is not not explanation," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 11–20. [Online]. Available: https://www.aclweb.org/anthology/D19-1002

[157] S. Serrano and N. A. Smith, "Is attention interpretable?" in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 2931–2951. [Online]. Available: https://www.aclweb.org/anthology/P19-1282

[158] S. Jain and B. C. Wallace, "Attention is not explanation," *CoRR*, vol. abs/1902.10186, 2019. [Online]. Available: http://arxiv.org/abs/1902.10186

[159] Y. Tay *et al.*, “Long range arena : A benchmark for efficient transformers,” in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=qVyeW-grC2k

[160] M. Ott *et al.*, “Scaling neural machine translation,” in *Proceedings of the Third Conference on Machine Translation: Research Papers.* Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 1–9. [Online]. Available: https://www.aclweb.org/anthology/W18-6301

[161] A. Buluc and J. R. Gilbert, “Challenges and advances in parallel sparse matrix-matrix multiplication,” in *2008 37th International Conference on Parallel Processing*, 2008, pp. 503–510.

[162] C.-Z. A. Huang *et al.*, “Music transformer,” in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=rJe4ShAcF7

[163] P. Shaw, J. Uszkoreit, and A. Vaswani, “Self-attention with relative position representations,” in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers).* New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 464–468. [Online]. Available: https://www.aclweb.org/anthology/N18-2074

[164] A. Luccioni, A. Lacoste, and V. Schmidt, “Estimating carbon emissions of artificial intelligence [opinion],” *IEEE Technology and Society Magazine*, vol. 39, no. 2, pp. 48–51, 2020.

[165] A. Radford *et al.*, “Language models are unsupervised multitask learners,” 2018. [Online]. Available: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

[166] A. Goyal *et al.*, “Recurrent independent mechanisms,” *CoRR*, vol. abs/1909.10893, 2019. [Online]. Available: http://arxiv.org/abs/1909.10893

[167] A. Lamb *et al.*, “Transformers with competitive ensembles of independent mechanisms,” *arXiv e-prints*, p. arXiv:2103.00336, Feb. 2021.

[168] F. Doray and M. Dagenais, “Diagnosing performance variations by comparing multi-level execution traces,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 462–474, Feb 2017.

[169] N. Ezzati-Jivan and M. Dagenais, "Multiscale navigation in large trace data," in *27th Annual IEEE Canadian Conference on Electrical and Computer Engineering (CCECE),2014*, 2014, pp. 1–6.

[170] B. Cornelissen *et al.*, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[171] F. Giraldeau *et al.*, "Recovering system metrics from kernel trace," in *OLS (Ottawa Linux Symposium) 2011*, June 2011, pp. 109–116.

[172] A. Hamou-Lhadj *et al.*, "Software behaviour correlation in a redundant and diverse environment using the concept of trace abstraction," in *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, ser. RACS '13. New York, NY, USA: ACM, 2013, pp. 328–335. [Online]. Available: http://doi.acm.org/10.1145/2513228.2513305

[173] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, Nov 2018.

[174] A. Benbachir *et al.*, "Automated performance deviation detection across software versions releases," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2017, pp. 450–457.

[175] D. Yuan *et al.*, "Context-based online configuration-error detection," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 28–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002181.2002209

[176] H. Nemati, S. V. Azhari, and M. R. Dagenais, "Host hypervisor trace mining for virtual machine workload characterization," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, 2019, pp. 102–112.

[177] M. Desnoyers and M. Dagenais, "Lttng: Tracing across execution layers, from the hypervisor to user-space," in *Linux Symposium*, 2008, p. 101.

[178] F. Giraldeau and M. Dagenais, "Wait analysis of distributed systems using kernel tracing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2450–2461, Aug 2016.

[179] M. Ester *et al.*, "A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise," in

Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, ser. KDD'96. AAAI Press, 1996, pp. 226–231. [Online]. Available: http://dl.acm.org/citation.cfm?id=3001460.3001507

[180] S. S. Skiena, *The Data Science Design Manual*, 1st ed. Springer Publishing Company, Incorporated, 2017.

[181] J. B. Tenenbaum, V. de Silva, and J. C. Langford, "A global geometric framework for nonlinear dimensionality reduction," *Science*, vol. 290, no. 5500, p. 2319, 2000.

[182] S. Forrest *et al.*, "A sense of self for unix processes," in *Proceedings 1996 IEEE Symposium on Security and Privacy*, 1996, pp. 120–128.

[183] R. Lippmann *et al.*, "Evaluating intrusion detection systems: the 1998 darpa off-line intrusion detection evaluation," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2, 2000, pp. 12–26 vol.2.

[184] G. Creech and J. Hu, "Generation of a new ids test dataset: Time to retire the kdd collection," in *2013 IEEE Wireless Communications and Networking Conference (WCNC)*, 2013, pp. 4487–4492.

[185] R. Caruana, S. Lawrence, and C. L. Giles, "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping," in *NIPS*, 2000.

[186] K. Yaghmour and M. R. Dagenais, "Measuring and characterizing system behavior using kernel-level event logging," in *2000 USENIX Annual Technical Conference (USENIX ATC 00)*. San Diego, CA: USENIX Association, Jun. 2000. [Online]. Available: https://www.usenix.org/conference/2000-usenix-annual-technical-conference/measuring-and-characterizing-system-behavior

[187] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 6645–6649.

[188] A. Graves and N. Jaitly, "Towards end-to-end speech recognition with recurrent neural networks," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 32. Bejing, China: PMLR, 22–24 Jun 2014, pp. 1764–1772. [Online]. Available: https://proceedings.mlr.press/v32/graves14.html

[189] R. Kiros, R. Salakhutdinov, and R. Zemel, "Multimodal neural language models," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 32. Bejing, China: PMLR, 22–24 Jun 2014, pp. 595–603. [Online]. Available: https://proceedings.mlr.press/v32/kiros14.html

[190] Y. Bengio, P. Frasconi, and P. Simard, "The problem of learning long-term dependencies in recurrent networks," in *IEEE International Conference on Neural Networks*, 1993, pp. 1183–1188 vol.3.

[191] Q. Fournier, G. M. Caron, and D. Aloise, "A practical survey on faster and lighter transformers," 2021. [Online]. Available: https://arxiv.org/abs/2103.14636

[192] Y. A. LeCun *et al.*, *Efficient BackProp.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 9–48. [Online]. Available: https://doi.org/10.1007/978-3-642-35289-8_3

[193] R. Thoppilan *et al.*, "Lamda: Language models for dialog applications," 2022. [Online]. Available: https://arxiv.org/abs/2201.08239

[194] C. Raffel *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[195] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/jmlr/jmlr13.html#BergstraB12

[196] N. Shazeer, "Glu variants improve transformer," 2020. [Online]. Available: https://arxiv.org/abs/2002.05202

[197] K. H. Zou, A. J. O'Malley, and L. Mauri, "Receiver-operating characteristic analysis for evaluating diagnostic tests and predictive models," *Circulation*, vol. 115, no. 5, pp. 654–657, 2007.

[198] N. Carlini *et al.*, "Extracting training data from large language models," 2020. [Online]. Available: https://arxiv.org/abs/2012.07805

[199] A. Holtzman *et al.*, "The curious case of neural text degeneration," 2019. [Online]. Available: https://arxiv.org/abs/1904.09751

[200] M. Tsimpoukelli *et al.*, "Multimodal few-shot learning with frozen language models," *Proc. Neural Information Processing Systems*, 2021.

[201] J. Haab, N. Deutschmann, and M. R. Martínez, "Is attention interpretation? a quantitative assessment on sets," 2022. [Online]. Available: https://arxiv.org/abs/2207.13018

[202] P. Foret *et al.*, "Sharpness-aware minimization for efficiently improving generalization," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=6Tm1mposlrM

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

[204] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[205] Y. E. Wang, G.-Y. Wei, and D. Brooks, "Benchmarking tpu, gpu, and cpu platforms for deep learning," 2019.

[206] Y. Wang *et al.*, "Benchmarking the performance and energy efficiency of ai accelerators for ai training," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 744–751.

[207] N. Nangia and S. R. Bowman, "Listops: A diagnostic dataset for latent tree learning," in *NAACL*, 2018.

[208] S. Narang *et al.*, "Do transformer modifications transfer across implementations and applications?" *arXiv e-prints*, p. arXiv:2102.11972, Feb. 2021.

[209] A. Wang *et al.*, "Superglue: A stickier benchmark for general-purpose language understanding systems," in *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/4496bf24afe7fab6f046bf4923da8de6-Paper.pdf

[210] S. Narayan, S. B. Cohen, and M. Lapata, "Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing.* Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 1797–1807. [Online]. Available: https://www.aclweb.org/anthology/D18-1206

[211] J. Berant *et al.*, "Semantic parsing on Freebase from question-answer pairs," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing.* Seattle, Washington, USA: Association for Computational Linguistics, Oct. 2013, pp. 1533–1544. [Online]. Available: https://www.aclweb.org/anthology/D13-1160

[212] I. Bello, "Lambdanetworks: Modeling long-range interactions without attention," in *ICLR.* OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=xTJEN-ggl1b

APPENDIX A SYSTEM CALL SEQUENCE OF A SIMPLE COMMAND

The following system calls sequence was generated by the execution of the command `echo "Hello World"` and collected by the `strace` tool. Note that `strace` only displays the first few arguments and interprets the raw value of the arguments to ease the reading.

Table A.1 System calls sequence corresponding to the execution of the `echo "Hello World"` command and collected by `strace`.

System call name	Arguments	Return value
execve	"/usr/bin/echo", ["echo", "Hello World"], 0x7ffe615a2e48 /* 24 vars */	0
brk	NULL	0x5597cd90a000
arch_prctl	0x3001 /* ARCH_??? */, 0x7ffe1620e090	-1 EINVAL (Invalid argument)
mmap	NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0	0x7f5aaba0b000
access	"/etc/ld.so.preload", R_OK	-1 ENOENT (No such file or directory)
openat	AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC	3
newfstatat	3, "", st_mode=S_IFREG|0644, st_size=60603, ..., AT_EMPTY_PATH	0
mmap	NULL, 60603, PROT_READ, MAP_PRIVATE, 3, 0	0x7f5aab9fc000
close	3	0
openat	AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC	3
read	3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"..., 832	832
pread64	3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64	784
pread64	3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848	48
pread64	3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0i8\235HZ\227\223\333\350s\360\352,\223\340."..., 68, 896	68
newfstatat	3, "", st_mode=S_IFREG|0644, st_size=2216304, ..., AT_EMPTY_PATH	0
pread64	3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64	784
mmap	NULL, 2260560, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0	0x7f5aab7d4000
mmap	0x7f5aab7fc000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000	0x7f5aab7fc000
mmap	0x7f5aab991000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000	0x7f5aab991000
mmap	0x7f5aab9e9000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x214000	0x7f5aab9e9000
mmap	0x7f5aab9ef000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0	0x7f5aab9ef000
close	3	0
mmap	NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0	0x7f5aab7d1000
arch_prctl	ARCH_SET_FS, 0x7f5aab7d1740	0
set_tid_address	0x7f5aab7d1a10	146836
set_robust_list	0x7f5aab7d1a20, 24	0
rseq	0x7f5aab7d20e0, 0x20, 0, 0x53053053	0
mprotect	0x7f5aab9e9000, 16384, PROT_READ	0
mprotect	0x5597cce70000, 4096, PROT_READ	0
mprotect	0x7f5aaba45000, 8192, PROT_READ	0
prlimit64	0, RLIMIT_STACK, NULL, rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY	0
munmap	0x7f5aab9fc000, 60603	0
getrandom	"\xa9\x6c\x39\xbe\xff\xb3\xb0\x8e", 8, GRND_NONBLOCK	8
brk	NULL	0x5597cd90a000
brk	0x5597cd92b000	0x5597cd92b000
openat	AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC	3
newfstatat	3, "", st_mode=S_IFREG|0644, st_size=5712208, ..., AT_EMPTY_PATH	0
mmap	NULL, 5712208, PROT_READ, MAP_PRIVATE, 3, 0	0x7f5aab25e000
close	3	0
newfstatat	1, "", st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ..., AT_EMPTY_PATH	0
write	1, "Hello World\n", 12Hello World	12
close	1	0
close	2	0
exit_group	0	?

APPENDIX B INTRODUCTION TO MACHINE LEARNING

At the dawn of artificial intelligence (AI), researchers rapidly tackled and solved problems that were challenging for humans but relatively straightforward for computers in that they could be described as a set of rules. Chess may certainly be the epitome of such complex tasks solved brilliantly by artificial intelligence. Nonetheless, despite the achievements of AI, simpler tasks that humans solve instinctively proved to be much more challenging as they are not easily expressed formally. Amongst others, speech and object recognition were – and still are to some extent – challenging problems to solve for computers. Machine learning (ML) provides a solution to intuitive problems by allowing computers to learn from experience instead of relying on human knowledge to specify the tasks or their solution. The seemingly ever-increasing amount of data produced every day has enabled machine learning to become suitable and successful for a wide range of simple and complex problems. Since the renaissance of deep learning (DL) associated with greedy layer-wise pre-training, neural networks have become the most popular family of algorithms for machine learning as they learn a hierarchy of concepts, with each concept defined through its relation to simpler concepts. As of the writing of this survey, deep learning, and more generally artificial intelligence, has become a thriving field with numerous practical applications that directly impact countless human lives, from medical diagnoses to movie recommendations.

APPENDIX C PRACTICAL GUIDELINES - GENERAL METHODS

The general approaches presented in Section 3.3 apply to the original Transformer as well as its lower-complexity alternatives. Therefore, they are discussed before introducing the specialized approaches in Section 3.4. In particular, this section provides practitioners and researchers with a series of guidelines on which methods to apply depending on the bottleneck and whether it occurs during optimization or inference. The distinction between optimization (i.e. pre-training and training) and inference is motivated by the former being significantly more resource-intensive than the latter.

The primary focus of this survey is to make Transformers more efficient and ultimately more affordable. Therefore, only substantial performance losses will be mentioned, along with other significant drawbacks such as incompatibilities and instabilities. Unless specified otherwise, the methods are readily available in PyTorch [203] and Tensorflow [204], two standard deep-learning libraries.

C.1 Optimization

Optimization is the most resource-intensive phase, prominently due to the iterative nature of the process, the quadratic complexity of the attention mechanism, and the in-memory recording of intermediate values during the forward pass. Consequently, most of the above approaches to reduce computation, memory, or both, focus on optimization.

C.1.1 Computation Savings

Recently, the undeniable success of pre-trained Transformers such as BERT [51], ViT [53], and GPT-3 [50] has confirmed the benefits of unsupervised pre-training. As previously mentioned, pre-training initializes the network's weights in a "good" region of space that allows the model to converge faster. Therefore, we advise practitioners and researchers to build upon pre-trained models like the ones available on the open-source library Hugging Face [112].

Nonetheless, pre-trained models are typically only available for "conventional" data and tasks such as translation, summarization, question answering, text-to-speech, image classification, and object detection. As for data and tasks without pre-trained models, we recommend initializing the model with a principled strategy such as Admin or T-Fixup and using a sample-efficient objective. Those techniques are not yet implemented in standard libraries, therfore we suggest using T-Fixup as it is simpler than Admin.

C.1.2 Memory Savings

As discussed before, although time may limit one's experiments, memory bottlenecks are much more critical. Since the intermediates values are responsible for a substantial part of the memory footprint, the first method to apply whenever memory is the main limiting factor during optimization is gradient checkpointing. The approach has two significant advantages: (i) the trade-off between memory and computation controlled by the number of intermediate values kept in memory is highly adjustable, and (ii) the method is straightforward to use in TensorFlow[1] and PyTorch[2]. Nevertheless, gradient checkpointing has some caveats with multiple GPUs, even on a single machine. For instance, as of the writing of this survey, gradient checkpointing interferes with PyTorch's Distributed and Data Parallel API, leading to instabilities[3].

Alternatively to gradient checkpointing, reversible layers provide a mechanism to recompute the intermediate values during the backward pass, thereby decoupling the model's depth from the amount of memory required by the activations. Although the increase in computation is reasonable, reversible layers produce numerical errors that may accumulate layers after layers to the point that they become an issue. Additionally, reversible layers are not yet part of standard libraries and require manually writing the forward and backward operations.

In addition to gradient checkpointing or reversible layers, parameter sharing allows further reducing the memory and is straightforward. However, unlike the other approaches, parameter sharing reduces the model's capacity. Fortunately, the trade-off between capacity and memory/computation savings is highly customizable, depending on the number of parameters shared.

Finally, a mixture of experts potentially with micro batching is expected to allow many memory-limited GPUs to train a Transformer even if each GPU is individually too small. However, both approaches require substantial effort to implement and impose a communication cost.

C.2 Inference

Sometimes, researchers have the resources to train large models during the development phase due to public or academic infrastructures, but they do not have the resources to deploy them. In such cases, one may do a neural architecture search to find the best model

[1] `https://github.com/cybertronai/gradient-checkpointing`
[2] `https://pytorch.org/docs/stable/checkpoint.html`
[3] `https://discuss.pytorch.org/t/ddp-and-gradient-checkpointing/132244/2`

within a parameter budget during training, preferably with So et al. [126]'s approach. As of this survey's writing, neural architecture search is not part of standard libraries.

Alternatively or additionally to NAS, structured pruning and distillation reduce the amount of memory and computations with fine-grained control. While structured pruning is already implemented, distillation is as easy as building a second model that predicts the teacher's output. As the aforementioned results suggest [90, 91, 96, 97, 98], the Transformer's performance does not significantly degrade when the model is pruned or distilled. Therefore, to reduce the amount of energy consumed by the model, we suggest applying those methods even when resources are sufficient during inference.

C.3 Optimization and Inference

The first and foremost method for faster and lighter models is automatic mixed-precision. Mixed-precision is compatible with virtually every neural network, combines with every other approach, reduces the memory footprint and accelerates computations on modern GPUs. Additionally, this method is one of the simplest to implement, only requiring a few lines of code in PyTorch[4] and TensorFlow[5].

Although 8-bit quantization may seem similar to 16-bit mixed-precision, the former is primarily used to speed up inference and is not as readily available as the latter. In particular, PyTorch does not provide quantized operators for GPU as of the writing of this survey, and Tensorflow warns users that "*different hardware may have preferences and restrictions that may cause slight deviations when implementing the spec that result in implementations that are not bit-exact*"[6]. Due to the finicky nature of 8-bit quantization, we suggest reserving this approach to specific hardware and use-cases such as the mobile setting.

[4]`https://pytorch.org/tutorials/recipes/recipes/amp_recipe.html`
[5]`https://www.tensorflow.org/guide/mixed_precision`
[6]`https://www.tensorflow.org/lite/performance/quantization_spec#specification_summary`

APPENDIX D PRACTICAL GUIDELINES - SPECIALIZED METHODS

With limitations mentioned in Section 3.5 in mind, let us examine the results of [159] and draw some broad guidelines.

The first observation is that every model is lighter than the original Transformer. Nonetheless, for memory-limited environments, the Synthesizer is the least advisable alternative as the model only reduces the memory by 24 to 26% regardless of the sequence length, which is consistent with its quadratic complexity. Instead, the Linformer, Performer, and Linear Transformer are better suited to address memory bottlenecks as they are at least 56% and 88% lighter than the original Transformer for input sequences of 1,000 and 4,000 tokens, respectively, which is also consistent with their linear complexity.

The second observation is that, on TPU V3 chips, the Synthesizer, the Reformer and BigBird perform roughly the same number of steps per second as the original Transformer regardless of the sequence length. In contrast, the Linformer, Performer, Sinkhorn Transformer and Linear Transformer are significantly faster than the original Transformer for input sequences of 4,000 tokens while performing on par for sequences of 1,000 tokens. Consequently, those models are better suited for computation-limited environments. We do not wish to overstate our claims here since TPUs and GPUs differ on some key aspects[1], and speed-ups may significantly vary, as observed by Wang et al. [205] and Wang et al. [206]. Although the data processing pipeline and the model implementation are outside this survey's scope, they should be tuned for the exact hardware used as it may significantly impact the performance.

Nonetheless, it would seem that the Linformer, Performer, and Linear Transformer are excellent options to improve memory and computation, with the Linformer standing out considering the simplicity of its implementation. However, those models also have serious drawbacks. The Linformer requires instantiating the projection matrices $\boldsymbol{E}$ and $\boldsymbol{F}$ of dimension $k \times n$, and thus can only process fixed-sized input sequences. Therefore, sequences must be padded to the size of the largest one in the dataset, which may significantly degrade the model's efficiency. The Performer and Linear Transformer are challenging to be efficiently implemented. Besides, they perform noticeably worse than the original Transformer on average. In some cases, such as byte-level text classification, they manage to outperform the original Transformer. In other cases, however, they might critically underperform. For instance, in

[1]Compared to modern GPUs with Tensor Cores, TPUs typically perform more FLOPs but have a lower memory bandwidth, have fewer but larger tiles, and apply the activation function within the matrix multiplication.

a longer variant of the ListOps task [207] that consist of modelling hierarchically structured data, they achieve less than 50% of the original Transformer's performance.

In contrast, sparse Transformers suffer less performance degradation on average, as measured on the Long-Range Arena benchmark. Notably, the LongFormer and BigBird achieved the same accuracy as the original Transformer for the ListOps task. Sparse models have, however, two major shortcomings. First, the sparsity must be structured in order to be efficiently implemented and yield practical improvements. Otherwise, the sparse model may be slower than its dense equivalent. Furthermore, CUDA kernels require considerable effort to be efficiently implemented and are specific to GPUs. Implementing equivalent kernels on TPUs is challenging, or even impossible, due to the disparity in supported primitives and operations. Secondly, dependencies that must be modelled to solve the task accurately should not be masked. Otherwise, the performance will be critically impacted. To select the appropriate sparse model, we recommend that one train a small vanilla Transformer with mixed-precision and gradient checkpointing, and then analyze the activation patterns of each layer's attention.

Nonetheless, in a recent paper, Narang et al. [208] investigated the impact of numerous modifications to the Transformer architecture, including changes in activation, normalization, depth, embeddings, and Softmax, on three NLP benchmarks, namely SuperGLUE [209], XSum [210], and WebQ [211]. The authors also evaluated several methods studied in this paper, including parameter sharing, Synthesizers, the Switch Transformer, and the Universal Transformer. They observed that no modification was able to improve the performance significantly. After ruling out various explanations, the authors conjectured that "*modifications to the Transformer architecture often do not transfer across implementations and applications*", which may explain why no modification has been widely adopted.

In conclusion, there seem to be no simple and universal guidelines regarding the current Transformer alternatives. If the data and task are standard, we recommend looking in the literature or on the Papers With Code website for references on how the different methods compare and experiment with already pre-trained models. Otherwise, we recommend using a small vanilla Transformer with mixed-precision and gradient checkpointing as baseline, then experimenting with already implemented lower-complexity models. As a side note, one may also want to combine multiple specialized approaches. For instance, BigBird-ETC relies on additional tokens for global attention, a form of memory similar to the Compressive Transformer. Nonetheless, many combinations are unprincipled at best. For instance, one should not factorize a sparse attention: the complexity will be similar to that of the same factorization of the full attention, and the sparsity may lose valuable information that the factorization could have preserved.

APPENDIX E ALTERNATIVES TO SELF-ATTENTION

Recently, attention-free alternatives to the Transformer have been proposed, putting Vaswani et al. [43] original paper title *Attention Is All You Need* to the test. Such architectures have not been explored in the core of this survey as they arguably remove the Transformer's core mechanism. Nonetheless, it is important to mention some of the most popular and promising alternatives.

Tolstikhin et al. [72] argued that self-attention is not required for image classification. They introduced a model called MLP-Mixer solely based on a succession of two multilayer perceptrons applied independently to image patches and channels, respectively, which achieved comparable accuracy to the ViT [53] on ImageNet.

Likewise, Liu et al. [73] argued that self-attention is not critical for computer vision and language modelling. They introduced a network called gMLP that models the interactions with Spatial Gating Units (SGU) instead of self-attention. Their model achieved the same accuracy as the ViT [53] on ImageNet, and the same perplexity of BERT [51] on a subset of C4.

Alternatively, Bello [212] proposed to replace the Transformer's self-attention with Lambda layers. Long-range content and position-based interactions are captured by transforming the context into linear functions, i.e. matrices, and applying them to each input independently. LambdaNetworks achieved comparable results to relatively small Transformers on ImageNet classification. While the memory complexity of Lambda layers remains quadratic with respect to the sequence length, it does not scale with the batch size. Additionally, the author proposed a multi-query variant that scales down the complexity by a factor.

Finally, Yu et al. [74] argued that the architecture of the Transformer is more valuable to the performance than the specific mechanism to relate the tokens. To illustrate their claim, the authors introduced the PoolFormer, a network that performs similarly to vision Transformers while replacing the self-attention mechanism with pooling, a simple non-parametric operator. Furthermore, the authors expanded on this idea with a more general and flexible architecture called MetaFormer, where the mechanism to relate the tokens is not specified while the other components are kept the same as the Transformer.

APPENDIX F SUMMARY OF THE SPECIALIZED APPROACHES

Table F.1 Summary of the specialized methods and their associated models.

Category	Approach	Model
Sparse	Fixed and Random Patterns	Star-Transformer [137]
		Sparse Transformer [138]
		Cascade Transformer [139]
		LogSparse-Transformer [140]
		BlockBERT [141]
		Longformer [142]
		BigBird [58]
	Learned and Adaptive Patterns	Sinkhorn Transformer [143]
		SparseBERT [144]
		Adaptively Sparse Transformer [145]
	Clustering and Locality-Sensitive Hashing	Reformer [87]
		Routing Transformer [148]
Factorized Attention	Low-Rank Factorization	Linformer [88]
		Synthesizers [149]
		Nyströmformer [150]
	Kernel Attention	Linear Transformer [152]
		Performer [151]
	Clustering and Locality-Sensitive Hashing	Transformer with clustered attention [153]
Architectural Change	Memory	Transformer-X Dai et al. [48]
		Compressive Transformer Rae et al. [154]
	Sequence Compression	Funnel-Transformer Dai et al. [155]