2024-06-03

Flow Size Prediction With Short Time Gaps

Hosseini, Seyed Morteza

UNIVERSITY OF CALGARY

Flow Size Prediction With Short Time Gaps

by

Seyed Morteza Hosseini

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

JUNE, 2024

Abstract

Having a priori knowledge about network flow sizes is invaluable in network traffic control. Previous efforts on estimating flow sizes have focused on long flows, where each flow is identified by a large time gap in the sequence of packets. However, many network control mechanisms such as load balancing and rate control achieve better performance when operating over flowlets, short flows that are separated by small time gaps in the sequence of packets. In this work, using extensive measurements, we investigate the feasibility of predicting the size of short flows, where the flow duration can be in the order of microseconds. Specifically, we deploy several popular workloads in a public cloud testbed, and collect both network and host traces for each workload. The network trace contains standard packet metadata, while the host trace contains high-level host statistics (*e.g.*,memory usage and disk I/O) and low-level function call traces (*e.g.*,*malloc()*, *send()*) that are captured during the execution of each workload via host instrumentation using eBPF. These traces are then used to train machine learning models for flow size prediction with varying time gaps ranging from microseconds to milliseconds. Our results indicate that:

(1) It is feasible to predict short flow sizes with high accuracy, *i.e.*, percentage error in 0-12% range,

(2) the low-level traces lead to 10-20% improvement in prediction accuracy compared to using the network and high-level traces.

Acknowledgements

I would like to express my profound gratitude to *Professor Majid Ghaderi* for his invaluable supervision, guidance, and support throughout the course of this research. His insights and expertise have been instrumental in the successful completion of this thesis.

I am deeply thankful to my family for their unwavering love and encouragement. To my mother, *Farbia*, and my father, *Alireza*, for their endless support and belief in my abilities; and to my sister, *Mehrnegar*, for her constant encouragement and companionship throughout this journey.

Additionally, I would like to extend my thanks to all my friends who have been there for me through this challenging yet rewarding process. Their support and camaraderie have made this journey not only possible but also enjoyable.

Each of you has contributed to my growth and this work in invaluable ways, and for that, I am eternally grateful.

Table of Contents

List of Figures

List of Tables

List of Symbols, Abbreviations, and Nomenclature

Symbol	Definition
SVM	Support Vector Machine
CNN	Convolutional Nueral Network
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative

Chapter 1

Introduction

1.1 Thesis Motivation

Many network control functions such as traffic optimization and resource allocation operate on traffic *flows* [1, 2]. A Flow is a set of packets that share the same 5-tuple of header fields (*i.e.*,source IP, destination IP, source port, destination port, and protocol), and is separated from the next set of packets by a *flow time gap*. Flow sizes depend on the flow time gaps, with shorter time gaps resulting in shorter flows and vice versa.

Accurate estimation of flow sizes is highly beneficial for network control functions that operate on flows. For example, in flow-based load balancing, having a priori knowledge of flow sizes allows the load balancing algorithm to more efficiently distribute traffic over the network links, thereby minimizing congestion in the network compared to a flow size-oblivious algorithm such as ECMP [3].

Previous studies [2, 4, 5, 1] leveraged the knowledge of the future flow size however they used simple and naive heuristics to model size of the next generated flows by the hosts

and the hosts network usage. For instance, systems like Helios [2] and FastPass [1] utilize the number of active flows as a proxy for estimating the bandwidth demand of the hosts. Other examples are pFabric [4] and Homa [5] data center transport protocols which rely on end hosts to embed the remaining size of the flow in their messages headers. However, the estimation methods employed by these systems, do not accurately predict the next flow size. Low accuracy flow size predictions can lead to less precise resource allocation and potentially undermine the overall efficiency of data center operations.

There is substantial research on predicting the volume of traffic transmitted in a given time interval using statistical techniques [6, 7] and deep learning models [8, 9]. However, much of this research is focused on predicting the aggregate traffic volume over fixed large time intervals. In contrast, our work is concerned with estimating individual flow sizes, specifically predicting the size of the next flow generated by a host.

Our work is inspired by [10], which uses machine learning to predict flow sizes for long flows, where the flow time gap is on the order of milliseconds. However, many network control functions such as load balancing and rate control achieve better performance when operating over flowlets, short flows that are separated by small time gaps in the order of microseconds [3]. In cases of small flow time gaps, there are more short flows, while larger flow time gaps result in fewer but longer flows. In Figure 1.1, we have plotted the Cumulative Distribution Function (CDF) of flow sizes extracted using three distinct flow time gaps from the traffic generated by a host in a training workload for a deep learning based image recognition application implemented in PyTorch. The figure shows varying the flow time gap leads to different flow size distributions.

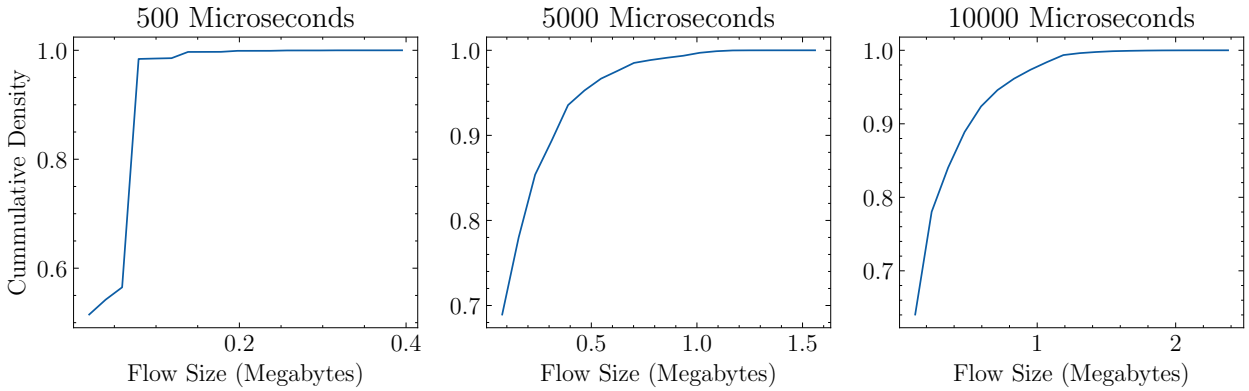


Figure 1.1: CDF of flow size for sample flow time gaps (500µs, 5000µs, 10000µs) of a deep learning training workload.

1.2 Thesis Objectives

In this study, we aim to explore the possibility of predicting the next data flow at various time scales. We approach this challenge from several angles.

- **Predicting the next flow using host level traces.** In this work, we aim to predict next generated flow size by using the captured traces on the host and assess the potential impact of varying flow time gaps on feasibility of the prediction. Our extracted traces can be grouped in three different categories,

  (1) packet trace (*e.g.*,packet size, packet address, *etc.*),

  (2) polled system traces (consist of high level traces captured from the host such as disk writes and reads amount, memory usage, *etc.*),

  (3) and event based traces (consist of traces of the events related to the host or the workload running on the host).

  To capture these traces, our trace collector module records a comprehensive set of system traces from within the host. To collect the packet trace, the trace collector

captures the communicated packets at the host and extract the essential data from each packet (*e.g.*,packet size, source ip, flow size, *etc.*). The polled system traces are obtained by periodically polling the desired system statistics from the "/proc" directory in Linux, which stores various information related to the processes in several files (*e.g.*,/proc/[pid]/stat which stores information about the process). The event based traces are captures using eBPF technology in Linux. First the desired hooks on different events are defined. The events can vary from calling a system calls (*e.g.*,*sys_send*) to a userland function call (*e.g.*,*malloc* in *clib*). These hooks capture the time of that event and the value related to that event (*e.g.*,total transmitted size in *sys_send*). These tools offer a significant advantage by providing the desired traces without the need to instrument the application code. While methods exist for monitoring and extracting traces that require modifications to the application code [11], it is often impractical to assume that such access or modifications can be made by the datacenter operator. Consequently, our research centers on gathering traces that can be acquired from the operating system without any alterations to the application code. By using the traces extracted from the host, we model the application behaviour in the hope of predicting it's network activities, specifically, the generated flow size.

- **Using machine learning for predictive analysis.** In order predict the size of the next generated flow, we utilize a machine learning model based on Gradient Boosting Decision Trees. For each examined flow time gap, we generate a dataset consisting of flows produced during communication with the specified time gap. These flows are extracted from packet traces, and the features for each flow are derived from various

traces collected from the host. Then, foreach dataset, we train and test a seperate model.

- **Next Flow Classification.** As opposed to predict the exact flow size, sometimes it is adequate to determine whether the next flow size exceeds a specific threshold. This threshold can be determined by how much bandwidth the flow uses. When a flow occupies more than 10% of the bandwith [12], it is called an elephant flow, and in the other case it is called a mice flow. There are scenarios that knowing whether the next flow is mice or elephant is useful. These use cases include but not limited to packet switching, scheduling, *etc.* [2] To address this, we train the same model as the exact size prediction but with binary labels (elephant/mice).

- **Effect of the flow time gap.** Our study explores the feasibility of predicting flow sizes at short timescales, spanning from microseconds to milliseconds. To achieve this, we deployed our target distributed workloads in two settings: a cloud-based testbed and a local testbed. Subsequently, we gathered detailed network and system traces from the host systems and assessed our predictive model using the extracted flows from these traces. Flow extraction was conducted with varying time gaps. Our goal is to further develop the understanding of the possibilities and difficulties involved in accurately predicting the sizes of flow within shorter time gaps. Our results shows that the flow size prediction is possible with acceptable accuracy in all of our evaluated workloads across all flow time gaps (*i.e.*,$MAPE$ less than 0.05 in most of our experiments in Sparks workloads and less than 0.12 for the Deep Learning Workloads). However, the prediction accuracy is influenced by the flow gap time. To understand this phe-

nomenon, we analysed the underlying factors responsible for the variance in predictive performance within these specific time gaps in a sample workload, the Pytorch model training.

- **Finding most impactful features.** To have a comprehensive understanding of the effect of different extracted features on flow size prediction, we conduct a comparison between a extended set of features and a limited set, in all of our evaluations. Through these evaluations, we identify the set of features that have more impact on prediction accuracy for certain workloads, for example the event-based features proven to be helpful in most of the spark workloads. Given that each captured trace imposes additional storage and computational overhead, acquiring an extensive set of traces for feature extraction may not always be feasible. Furthermore, a large feature set increases the risk of model overfitting. Therefore, identifying the most influential features is valuable in numerous scenarios.

1.3 Thesis Contributions

Our main contributions in this work are:

- We developed a data collector module designed to capture traces from the host using operating system tools (specifically Linux) without requiring any modifications to the workload code. The trace collector operates as a monitoring process alongside the application process during workload execution. It employs eBPF technology and integrates Linux monitoring tools, as well as *PF_RING*, to effectively capture packets

and extract detailed traces from each host involved in the workload. This enables the collection of various trace types across different workload categories, streamlining the experimental process.

- We conducted an extensive collection of low-level network traffic and host traces from five distinct workloads, which we deployed on a public cloud platform, specifically AWS, to benefit from its GPU-capable nodes. Our selection of workloads falls into two main categories: GPU-based and CPU-based. On the CPU front, we executed three well-known algorithms on Spark using the Scala language: SVM with Stochastic Gradient Descent, K-means clustering, and the PageRank algorithm. Alongside these, we trained a concise image recognition model using TensorFlow on CPU nodes and developed a scaled-down version of a GPT model with PyTorch, which we trained on GPU nodes. These diverse workloads provided a comprehensive dataset for our analysis.

- We introduced a novel set of features, referred to as event-based features, which are extracted from function calls during workload execution using *eBPF* technology. The scope of these traces spans various user space functions, such as memory allocation, as well as different kernel system calls like *read()* and *write()*. These fearures reflect the intricate micro-interactions between the operating system and the application and also different parts of the application and operating system within themselves. These feature can represent the characteristics of the workload, For instance, workloads that are I/O intensive demonstrate a high frequency of *read* and *write* system calls. Conversely, workloads that are computationally intensive, particularly those leveraging

GPU resources, are characterized by a significant number of memory allocations using functions such as *cudaMalloc*. Our extensive evaluation of these features has demonstrated their utility in enhancing the model's predictive performance.

- We systematically analyze the impact of flow time gap on the model accuracy. Our results indicate that

  (1) Predicting the size of short flows is feasible, as demonstrated by low Mean Absolute Percentage Error (MAPE), indicating a minimal percentage error in our predictions.

  (2) The inclusion of event-based features enhances the model's predictive accuracy. For instance, in Spark workloads, these event-based features have high importance score, highlighting their significance in improving model performance.

1.4 Thesis Organization

This thesis is organized as follows: Chapter 1 introduces the thesis's motivation, goals, and structure. Chapter 2 covers the necessary background on regression and classification trees, and the library we used for modeling, the Catboost. Additionally in includes the tools and methods used for trace collection such as eBPF and *PF_RING*. Chapter 2.2 represents the review of relevant recent studies. Chapter 3 discusses the workloads we utilized, describing the methodologies and tools applied. Chapter 4 details our trace collection tool and the predictive model used. In Chapter 5, we delve into the evaluation of our model. The thesis is concluded in Chapter 6.

Chapter 2

Background and Related Works

In this chapter, we lay the groundwork by exploring the fundamental concepts and methodologies that inform our research into network traffic prediction. This includes an overview of existing traffic and flow size prediction research, alongside the key tools and techniques such as eBPF, the Linux proc directory, *PF_RING*, and machine learning models like CatBoost and Gradient Boosted Trees. Through this exploration, we establish a basis for our investigation into enhancing the accuracy and efficiency of network flow size predictions.

2.1 Background

In this section, we explore the essential concepts that form the basis of our study.

2.1.1 Flow vs. Flowlet

In our study, the term "flow" refers to a set of packets that share the same 5-tuple header and are separated from the next set of packets by a specific flow time gap (Δt). In the

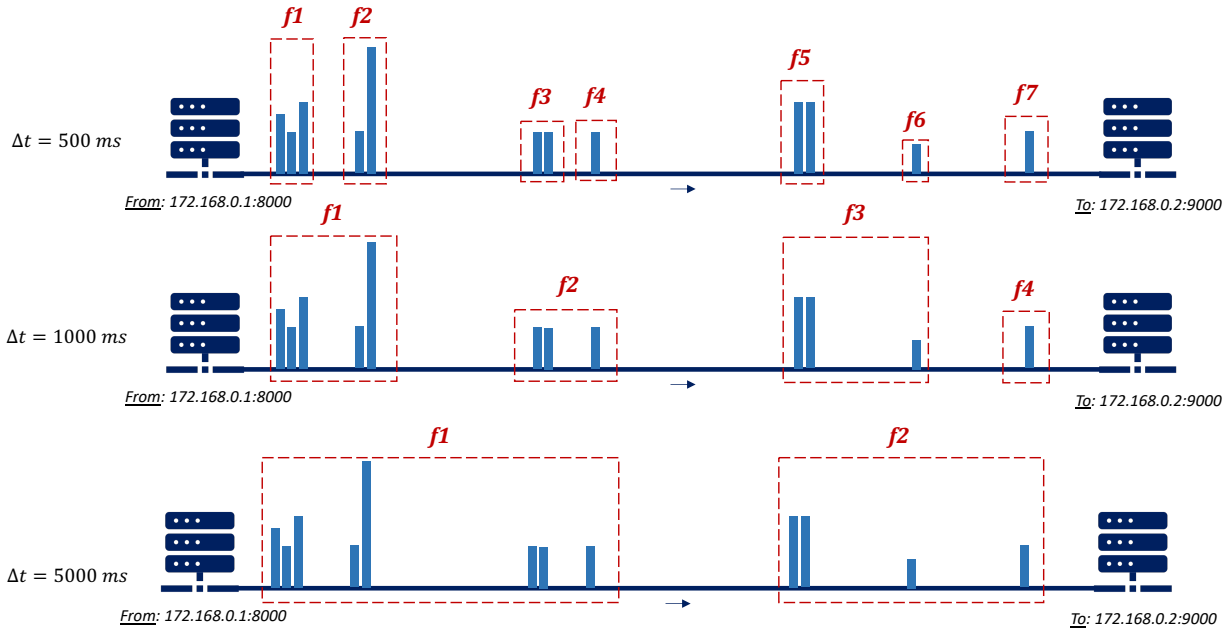


Figure 2.1: The effect of varying flow time gaps (Δt) on the structure of flowlets within the same network traffic. As Δt increases, the number of flowlets decreases, leading to larger, more aggregated flows. This demonstrates the challenge of predicting flow sizes accurately across different time gaps.

literature, this is often referred to as a "flowlet".

The diagram in Figure 2.1 illustrates how varying the flow time gap changes the flowlets within the same network traffic. When we adjust flow time gap, the definition of what constitutes a single flowlet changes. For example, with a flow time gap of $500ms$, the network traffic is divided into several smaller flowlets (f1, f2, f3, etc.). Increasing flow time gap to $1000ms$ or $5000ms$ results in fewer, larger flowlets because the packets are more likely to be grouped together.

This process highlights the challenge of short time gaps. With shorter flow time gaps values, there are many flowlets within a single flow, making it more complex to accurately predict the flow size. Conversely, longer flow time gaps values simplify the prediction task but may not capture the granularity required for some network management functions.

By capturing the traffic once and analyzing it with different flow time gaps, we can

observe how the flowlet structure changes. This helps us understand the impact of flow time gaps on the consistency and accuracy of our flow size predictions, addressing both short and long flow challenges.

2.1.2 Network Traffic Prediction

Network traffic prediction involves the use of various methods to forecast future network patterns. This predictive capability is crucial for maintaining network efficiency, optimizing resource allocation, enhancing user experience, and ensuring the stability and reliability of network services. Within the scope of this thesis, our focus narrows down to predicting the amount of traffic, divided into two specific domains: network volume prediction and flow prediction [13].

Network Volume Prediction

Network volume prediction aims to forecast the total volume of traffic that will traverse a network within a given timeframe. This entails predicting key metrics, such as the overall number of bytes, that are expected to be transmitted either throughout the entire network or within designated segments or channels.

Accurate traffic volume predictions are vital for effective network management. They allow network administrators to perform capacity planning, ensuring that the network can handle expected loads without degradation of service quality. Additionally, predictions help in dynamically adjusting resources to meet varying traffic demands, thus optimizing network performance and preventing congestion [13].

The main challenges in network volume prediction include the dynamic and unpredictable

nature of network traffic, influenced by user behavior, technological changes, and various external factors. Additionally, the accuracy of predictions can be affected by the granularity of the data and the time horizon of the forecast.

Time series analysis methods are used to detect patterns, trends, and cyclical behavior in network traffic volume by analyzing historical data. This analysis is helpful in predicting future traffic volumes by leveraging patterns observed in the past.

Flow Size Prediction

Flow size prediction in network traffic aims to predict the exact size of specific network flows. Each network flow is defined as a sequence of packets transferred between a source and destination, identifiable by attributes such as source and destination IP addresses, port numbers, and the protocol utilized. Effective flow size prediction can significantly enhance network management and planning. It enables network operators to optimize resource allocation, ensure quality of service (QoS) for critical applications, and implement efficient traffic shaping and prioritization strategies.

Contrary to volume prediction, which looks at the aggregate traffic, flow size prediction examines the network at a more granular level, focusing on the characteristics and behavior of specific flows. This method provides in-depth understanding of how specific services are used, identifies when these services are most in demand, and highlights possible congestion points within the network flow. The primary challenges in flow size prediction include the high variability of flow durations and sizes, the potential for rapid changes in network conditions, and the sheer volume of flows in large networks. Flow size prediction often employs statistical models, machine learning algorithms, and deep learning techniques. These models are trained

on historical flow data, including timestamps, packet sizes, and flow durations, to predict future flow size.

Elephant and Mice Flows

In the context of flow prediction, it is crucial not only to predict the exact size of the flow but also to classify the flow into two categories—specifically, as either an elephant flow or a mice flow. An elephant flow is defined as a flow that occupies more than 10% of the available bandwidth [12]. Conversely, flows that consume less bandwidth are classified as mice flows.

Classifying flows into these categories is particularly beneficial in scenarios such as packet switching and scheduling [2]. Understanding whether the next flow will be an elephant or a mice flow allows for more efficient network management and resource allocation. For instance, elephant flows, due to their substantial size, can cause congestion and require different handling strategies compared to mice flows, which are smaller and more numerous. By accurately predicting these elephant flows before they occur, network operators can better allocate bandwidth, ensuring these large transfers do not disrupt overall network performance. In contrast, mice flows are small and numerous. They often represent the majority of traffic by count but consume relatively little bandwidth compared to elephant flows. Efficient handling of mice flows is crucial for optimizing network resource usage without over-provisioning.

2.2 Related Works

This section provides an overview of the existing research in traffic and flow size prediction, highlighting the methodologies and approaches used in the field. It sets the stage for discussing the advancements in predicting network traffic dynamics, from heuristic methods to learning-based approaches, and positions our work within the broader context of flow size prediction.

2.2.1 Traffic Prediction

Traffic prediction is an important problem in network control, management, and planning. There is extensive research on traffic prediction using statistical [6, 7] and deep learning techniques [8, 9]. As a notable research [14] explores the use of Long Short-Term Memory (LSTM) neural networks for classifying and forecasting real-time server traffic flows in hybrid Electro-Optical (E/O) data center networks. Their study highlights the effectiveness of LSTMs in managing the classification of server traffic flows, characterized by their high variability and burstiness. Furthermore, they present promising outcomes in forecasting future traffic, demonstrating the LSTM's capability for accurate one-step and multi-step ahead predictions. The study focuses on improving network resource utilization and minimizing operational costs by implementing smart traffic management techniques. It explores how strategically managing network traffic can optimize the use of network resources, thereby reducing the financial burden on operational expenses. This approach leverages data-driven models to forecast and classify network behavior, enabling more efficient and cost-effective network operations.

However Traffic prediction researchs are focused on predicting the aggregate traffic volume over fixed long time intervals. In contrast, our work is concerned with estimating individual flow sizes, which is a more challenging problem as traffic aggregation often smooths out variability, which would improve prediction accuracy.

2.2.2 Flow Size Prediction

Various methodologies have been employed in the prediction of flow sizes. We can categorize these methods into two broader groups: learning-based methods and heuristic methods.

Heuristic Methods

Before the advent of learning-based approaches for flow size predictions, various network systems (*e.g.*,transport protocols, scheduling, *etc.*) defined flow prediction heuristics for their estimation modules. These heuristics encompassed a variety of techniques. Hedera's [12] flow demand estimation is designed based on TCP behavior (Fairness and AIMD) to predict the demand of ongoing flows. The pFabric [4] datacenter transport protocol modifies the network stack, embedding the remaining flow size in the packet header, and uses this information for flow scheduling. Mahout [15] monitors socket buffers of hosts to detect the ongoing flow size category (*i.e.*,Elephant/Mice) at the hosts. Many other works [1, 2, 5] have implemented different heuristics for flow size estimation and prediction.

Learning-Based Methods

One important donwside of the heuristic methods is that they only predict the size or category (*i.e.*,Elephant/Mice) of the ongoing flows. This limitation may lead to various issues, such as

congestion. Learning-based methods are introduced to mitigate these problems by predicting flow sizes in advance, either before or at the beginning of flow initiation.

[16] present a network management approach through the application of online flow size prediction techniques aimed at enhancing network routing, particularly within the context of data mining for computer networks. This work tackles the challenge of predicting flow sizes in real-time and investigates its adoption in routing, load balancing, and scheduling across network systems. By adopting an online machine learning framework, the research adapts dynamically to the evolving patterns of network traffic. Utilizing specific flow characteristics such as source and destination IPs, port numbers, protocols, and the size of initial packets, they employ neural networks, Gaussian process regression, and online Bayesian Moment Matching to classify upcomming flows. Their findings suggest significant possibilities for reducing the flow completion time and boosting the efficiency of routing by promptly identifying and managing "elephant" flows. This research provides insightful directions for the advancement of traffic management technologies.

FLUX [10] demonstrates the performance of learning-based methods when provided access to a broader array of features. Specifically, *FLUX* employs a machine learning technique, such as Gradient Boosting Trees [17], to predict the size of upcoming flows. The feature set of *FLUX* includes various statistics extracted from host machines and network communications (*e.g.*,memory usage, flow start time). Evaluating the impact of utilizing a learning-based method like *FLUX* across several scheduling systems revealed an enhancement in flow completion time (FCT) with knowledge of upcoming flow sizes.

Building on *FLUX*'s framework, [18] employs a random forest model, using a feature set simillar to *FLUX*, to classify upcoming network flows as elephant or mice. DarkTE incorpo-

rates a confidence-based system for rate allocation and path selection, effectively mitigating the impact of sporadic classification inaccuracies and thereby enhancing overall routing efficacy. This methodology increase the speed and precision of flow classification—achieving speeds in the hundreds of microseconds and accuracy rates of at least 86.4%—across a spectrum of realistic workload scenarios. Notably, DarkTE's optimization of flow completion times and its superior bandwidth utilization, when compared to conventional strategies like Hedera, underscore its significant potential to augment resource efficiency and operational productivity within data center environments.

As an instance of data center systems leveraging learning-based flow prediction, *LAFS* [19] is a method for flow scheduling aimed at reducing flow completion time (FCT) in data center networks. *LAFS* merges system call monitoring with machine learning techniques to precisely predict the size of data flows, enabling an optimized scheduling strategy based on the Shortest Remaining Processing Time (SRPT) principle and the strategic use of in-network priorities. Leveraging the machine learning framework established by *FLUX* and utilizing eBPF for system call monitoring, *LAFS* offers a significant improvement in network performance across various workloads without the need for altering application interfaces or deploying specialized hardware. Through different simulations it is illustrated that *LAFS* outperforms conventional designs that do not use advance flow information.

In our work, in contrast to previous learning-based methods, we utilize a more extensive feature set, incorporate event-based features into the model, and assess the impact of flow time gap on flow size prediction.

2.3 Tools and Techniques

In this section we delve into the tools and techniques we used in our research. We start with eBPF technology which play a critical role in tracking low-level system behavior. We then move on to the Linux proc directory and *PF_RING*, which are vital for monitoring network traffic and also system behavior. Our focus also extends to the machine learning approaches utilized in our research, particularly CatBoost and Gradient Boosted Trees. We wrap up the chapter with a summary of the metrics used to assess the accuracy and effectiveness of our predictive models, preparing the ground for an in-depth discussion of our research methods and outcomes.

2.3.1 Extended Berkeley Packet Filter

Extended Berkeley Packet (eBPF) Filter is a technology inside the Linux kernel that has transformed how networking, security, and performance monitoring are implemented at the system level. eBPF facilitates the execution of programs within the kernel space, eliminating the need to modify kernel source code or load additional modules. These programs can be written in a high-level C like language that is compiled into eBPF bytecode and run by the kernel's eBPF virtual machine. This feature enables developers to introduce new functionalities into the kernel dynamically, providing notable flexibility and options for customization.

eBPF Architecture

A key aspect of eBPF is its event-driven architecture. eBPF programs can be attached to a variety of hooks in the Linux kernel, such as system calls, network events, and kernel

tracepoints. This means that eBPF programs are triggered by specific events occurrences in the kernel, allowing them to run in response to particular conditions or activities. For example, an eBPF program can be triggered whenever a certain network packet is received, providing a powerful way to monitor and filter network traffic directly from the kernel.

Contrary to the standard user-space applications, eBPF programs are executed in a dedicated virtual machine inside the kernel space. This eBPF-specific virtual machine offers a stable and secure environment for executing operations, managing the processing of eBPF bytecode, and facilitating direct interaction with the kernel's internal resources. For example when tracing a system call such as `read` with eBPF, the eBPF code has access to the parameters passed to the `read` function.

eBPF Programs

eBPF programs are developed using a constrained subset of the C programming language. Compilation of these programs into eBPF bytecode is accomplished through the LLVM and Clang Toolchain [20]. Subsequently, the generated bytecode is loaded into the eBPF virtual machine via the `bpf` system call. Due to the potential security and stability concerns associated with code execution within the kernel, a series of verifications are conducted on each eBPF program prior to its loading. These verifications consist of ensuring program termination, examining for pointer boundary violations, verifying the integrity of registers and stack states, *etc.* After the verification, the in-kernel Just-in-Time (JIT) compiler processes the eBPF bytecode, converting it into machine code. This JIT compiler is designed to produce machine instructions during runtime, which are then allocated to executable memory for direct execution. This process allows the efficient execution of tailored instructions within

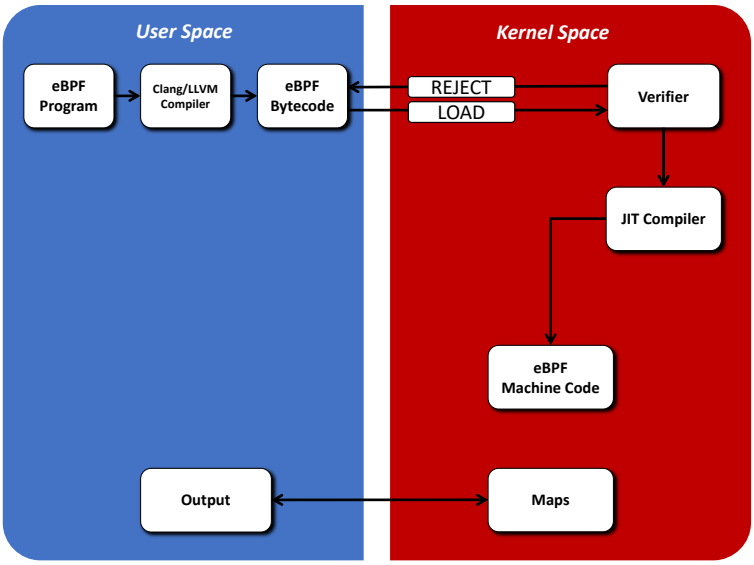


Figure 2.2: eBPF program loading and verification procedure.

the kernel environment. In Figure 2.2 the eBPF program loading and verification process is shown.

eBPF Maps

eBPF maps are a fundamental aspect of the eBPF framework, offering a practical means to store and exchange data between eBPF programs, as well as between the eBPF program in kernel and the loader program in user space. These maps are key-value stores in the kernel, enabling eBPF programs to maintain state, compile data, and interface with user space applications. The eBPF maps has serveral types including hash tables, arrays, and bloom filters. eBPF maps have diverse set of use-cases, from monitoring network connections to managing configuration data. Their shared accessibility among different eBPF programs and the kernel and user space enhances eBPF's functionality, facilitating sophisticated, stateful operations efficiently and securely.

eBPF Hooks

eBPF programs are triggered upon execution of specific hook points, either within the kernel or during an application's execution. These hook points may include a range of predetermined locations, such as Kernel tracepoints, network-related events, or system calls. Additionally, they can originate from custom kernel probes (kprobes) or user-space probes (uprobes). For example in our use-case, we save the sent messages sizes with `sendmsg` system call in an eBPF map in our eBPF code and collect the stored sizes in our user-space program.

BCC

BCC [21] is a toolchain which faciliates the creation of eBPF programs. With BCC, users can write Python user-space programs with embedded eBPF code inside. It offers tools for the generation of eBPF bytecode, its loading into the kernel, and interaction with the program via the eBPF Maps through a Python API. The bcc project also offers various predefined tools for tracing and monitoring of various aspects of the system through eBPF, such as disk I/O, memory, network events, *etc.*

2.3.2 Linux *proc* Directory

The */proc* directory in Linux acts as a virtual filesystem offering insights into the kernel's perspective on the operating system. It hosts virtual files that represent system and process information in real-time, such as memory allocation, mounted devices, and hardware setup. This special directory serves as a dynamic interface for accessing data about the system's

state and running processes without storing actual files on the disk.

Here we describe integral parts of the */proc* directory:

- **Process Information.** The */proc* directory in Linux provides a detailed view into each active process, represented by a subdirectory named after the process's ID (PID). Within these subdirectories, one can find various details about the processes, including their status, memory consumption, and open file descriptors, offering a comprehensive snapshot of their current operation. For example */proc/[PID]/status*, Shows the current status of the process, including its state (running, sleeping, etc.), memory usage, and various counters, and */proc/[PID]/stat*, Offers detailed statistics about the process, such as its PID, parent PID, CPU usage times, and more.

- **System Information.** The */proc* directory extends beyond individual processes, encompassing files that reveal comprehensive system metrics. For instance, */proc/cpuinfo* delivers specifics on the CPU's characteristics and capacity, while */proc/meminfo* delineates the memory's distribution and availability.

- **Virtual Files.** Files in the */proc* directory are virtual because they reside in memory and not on disk. Reading from these files reads directly from kernel data structures.

- **Interface to Kernel Data Structures.** The */proc* filesystem is an interface to internal data structures of the kernel. It allows users and applications to query the kernel for information about the system and its processes.

- **Sysctl Interface.** In the Linux */proc* filesystem, select files can be modified, enabling

alterations to the kernel's operational settings in real time. For instance, by writing to the */proc/sys/vm/drop_caches* path, one can command the kernel to clear specific cache types.

- **Dynamic and Real-time.** The contents of the */proc* filesystem are generated dynamically and provide real-time system information.

The */proc* directory serves as a crucial resource for tools that monitor a Linux system, offering direct insights into the status and performance of the system. It is also frequently utilized by system administrators for diagnostics and system optimization tasks.

2.3.3 PF_RING Library

PF_RING™ [22] is a library designed for high-speed packet capture and processing. PF_RING enhances packet capture efficiency via an optimized kernel module. This module facilitates rapid, low-level packet copying into specialized constructs known as PF_RING rings. Then, the user-space applications can retrieve packets from the PF_RING rings by using the PF_RING API.

PF_RING Architecture

PF_RING is composed of two parts:

1. A kernel module which defines and manages a ring buffer, and provides low-level copying of packets into this buffer.

2. A user-space API, delivered through an adapted version of libpcap [23] (*i.e.*, libpfring), which is a widely known library for packet parsing and processing.

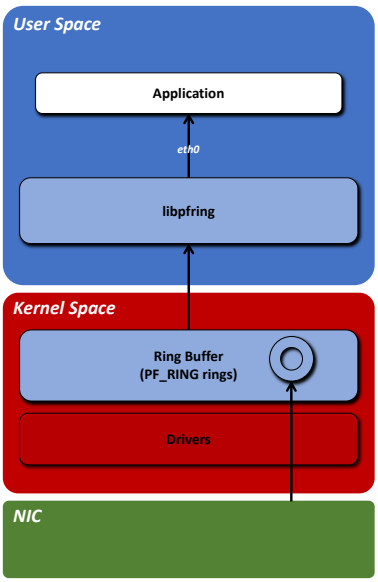


Figure 2.3: PF_RING Architecture.

Upon the arrival of a packet at the network interface controller (NIC), PF_RING transfers the packet into the ring buffer via the Linux Network API. User-space applications can access these incoming packets using the API provided by PF_RING. Notably, PF_RING optimizes performance by avoiding the allocation and deallocation of memory for each arriving packet. Instead, once a packet is processed from the ring, its space is reused for next packets. This operational characteristic implies that PF_RING is not designed to store packets for extended periods. Consequently, applications leveraging PF_RING need to store packets if long-term usage or analysis is required.

It is important to see that PF_RING defines a specialized socket type to facilitate the connection between the user-space API and the kernel. This socket type serves as a bridge between the kernel module and the user-space API, thereby enabling seamless interaction. In Figure 2.3 the overall architecture of the PF_RING is shown.

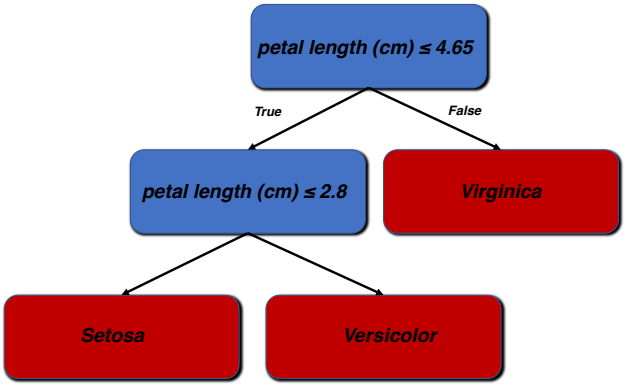


Figure 2.4: Structure of a simple classification tree for plant classification on Iris dataset.

2.3.4 Machine Learning Tools

Classification and Regression Trees

Classification and Regression Trees [24] constitute a versatile set of algorithms used for both classification and regression tasks in machine learning. Essentially, these algorithms iteratively divide the data into subsets based on certain criteria. Consequently, this partitioning process can be visually shown in the form of a decision tree. In classification trees, the goal is to partition the dataset into subsets that correspond to the different categories in the target variable. On the other hand, regression trees usually aim to divide the dataset in a way that minimizes the squared error of the continuous target variable within each subset. The simplicity and ease of interpretation of classification and regression trees algorithms, and their competence in managing data of both numerical and categorical types, make them a popular option for predictive modelling tasks. Figure 2.4 shows a simple classification tree structure for plant classification on Iris [25] dataset. We can see the split condition on non-leaf nodes (*e.g.*, petal length < 4.65 in root node) and the target class on each leaf node of the tree.

Advantages and Disadvantages

Understanding the strengths and limitations of classification and regression trees is crucial for a comprehensive analysis. We begin with the advantages:

- **Visualization.** Once constructed, the tree can be visualized, offering profound insights into the data. Moreover, it can be effortlessly transformed into a set of rules for making predictions.

- **Non-Parametric Nature.** Tree-based algorithms do not presuppose a specific relationship between independent and dependent variables (X and Y), unlike parametric models like deep learning models or linear regressions.

- **Speed.** These models are known for their computational efficiency and can be further optimized on specialized hardware such as FPGAs.

- **Interpretability.** Tree-based models generate a set of easily understandable rules for predictions. For instance, when assigning a class to a sample, it's clear which variables influence the decision-making path from the root to the leaf.

Despite these advantages, classification and regression trees also have their drawbacks:

- **Overfitting.** here's a tendency for these models to overfit, leading to low bias but high variance in predictions. This can hinder the model's ability to generalize to new, unseen data.

- **Error propagation.** Errors in higher-level decisions can cascade through the tree, as

each node's decision is based on the outcomes of previous nodes.

- **Sensitivity.** The model's reliance on conditions for splitting data at nodes means that slight changes in the training data can significantly alter the tree's structure.

Ensemble Learning

Ensemble Learning techniques have gained significant interest within the machine learning field over recent decades. These approaches are build on top of the concept of reducing variance to enhance the accuracy of decisions. Historically, ensemble methods boast a successful track record in various applications, including feature selection, confidence estimation, and managing class-imbalanced data, among others [26].

Ensemble Learning methods operate by training an array of distinct learners (*i.e.*, machine learning models) and combining their predictions, often through strategies like averaging. When these individual models are combined within an Ensemble Learning framework, the collective system generally exhibits enhanced generalization compared to each standalone model. To understand the cause of this improvement in generalization, it's crucial to acknowledge that these individual learners are inherently imperfect, each possessing a nonzero likelihood of error. The error in these learners is constituted of two elements: bias, which is the consistent deviation from the true value, and variance, reflecting the model's sensitivity to fluctuations in the training dataset. The primary objective of ensemble methods is to reduce the variance component in each learner's error by averaging the outputs of all learners. As averaging inherently reduces variance (*i.e.*, Smoothing effect), this objective is often successfully met, leading to a more robust and reliable prediction model [27].

A notable concern regarding ensemble methods is that they do not necessarily assure superior performance compared to the most effective individual learner within the ensemble. However, these methods do have the advantage of reducing the likelihood of relying on a learner that exhibits a poor performance among all learners [27].

Ensemble learning techniques can be broadly classified into two types, based on the generation of the individual learners. First, the Boosting technique, constructs individual learners that are strongly correlated and produces these learners in a sequential manner. Second, the Bagging technique, generates individual learners independently, allowing for the parallelization of the learner creation process.

Boosting

Boosting operates by iteratively constructing a series of models such as decision trees, with each subsequent model focusing on reducing the error of its predecessors. Typically, boosting ensemble methods initiate by assigning a set of weights to the data points in the training set, signifying the importance of each sample in the model construction process. A model is then trained on this weighted data, with the weights playing a crucial role in the training phase. For instance, in a classification tree, the weights influence the computation of measures like the Gini index. After the new learner is trained, the learner's error is computed and weighted according to the assigned weights of the data points. Then the ensemble system adjusts weights of the training data samples and reduces the weights of the instances that were predicted accurately. The final model is the weighted combination of all trained learners where the learner with lower error rates has a higher weight. In Figure 2.5 the overall process of the boosting algorithm is illustrated.

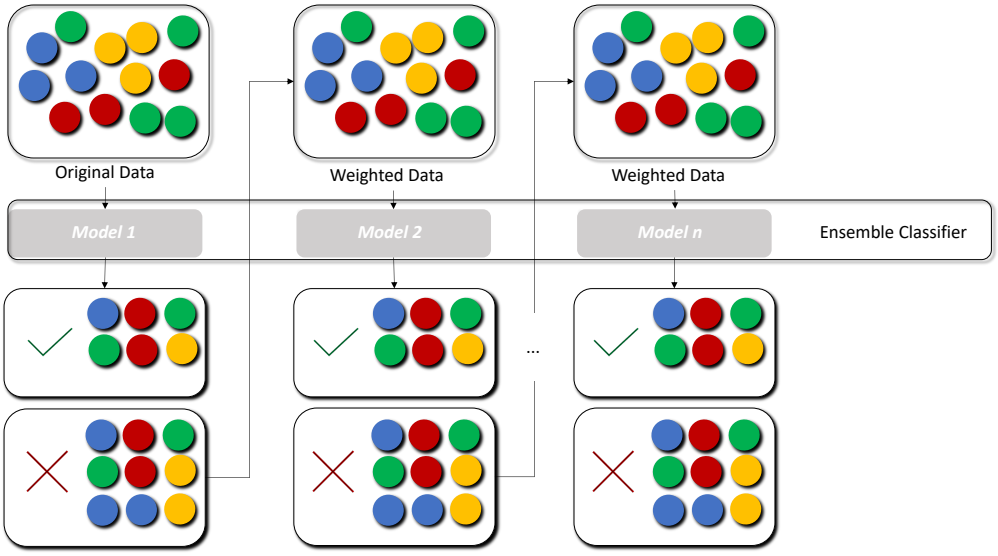


Figure 2.5: The overall architecture of boosting process.

The goal of boosting is to create a strong learner from an ensemble of learners. By focusing on correcting the misclassifications of previous models, boosting can adaptively improve and often results in a model with high accuracy. Popular boosting algorithms include AdaBoost [28] (Adaptive Boosting), Gradient Boosting [29], and XGBoost [30], each with its own way of building individual learners and combining them.

Gradient Boosting

Gradient Boosting [29], a technique built on top of the concept of Boosting, is employed for developing predictive models in a systematic, step-by-step manner. It refines the models by optimizing a differentiable loss function.

This method aims to approximate the function that maps input data X to the target variable Y. The approximation is a weighted aggregate of learners, constructed across multiple boosting rounds:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x) \tag{2.1}$$

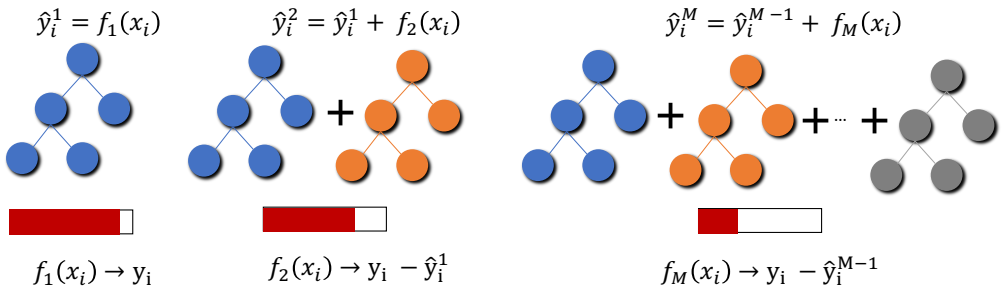


Figure 2.6: The overall procedure of gradient boosting process.

Here, γ_m represents the weight of the m^{th} learner, h_m. The process begins with an initial function, F_0, which serves as the starting point of the ensemble. For regression, this might be the mean of the target variable Y, and for classification, it could be the log odds ratio:

$$F_0(x) = \mathbb{E}[Y] \tag{2.2}$$

Subsequent to this initialization, the following objective is minimized:

$$(\gamma_m, h_m(x)) = argmin_{\gamma,h} \sum_{i=1}^{N} L(y_i, F_{m-1}(x_i) + \gamma h(x_i)) \tag{2.3}$$

Gradient descent is utilized for this minimization, where each learner h_m is trained on a new dataset $D = \{x_i, r_{im}\}_{i=1...N}$, with pseudo residuals r_{im} defined as:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \tag{2.4}$$

This resembles a single step in the gradient descent algorithm. Moreover, γ_m is determined by solving a line search optimization problem. The final model is given by:

$$F(x) = F_0(x) + \sum_{m=1}^{M} \eta \cdot h_m(x) \tag{2.5}$$

In Figure 2.6 the overall process of the gradient boosting algorithm is illustrated.

CatBoost

CatBoost [31] is a gradient boosting machine learning algorithm widely known for its effectiveness in handling categorical data, along with its efficiency, accuracy, and ease of use.

A key attribute of CatBoost is its innate capability to process categorical data directly, distinguishing it from other algorithms that typically necessitate extensive preprocessing to transform categorical data into a numerical format. CatBoost employs a efficient encoding strategy, similar to one-hot encoding but enhanced, to adeptly manage categorical variables.

Gradient boosting faces a challenge known as Prediction Shift, a problem CatBoost identifies and addresses. Prediction Shift occurs when the conditional distribution of the training data's gradient, $g^t(x_k, y_k)|x_k$, diverges from the gradient distribution of the test samples, $g^t(x, y)|x$. This discrepancy biases the learner, h^t, towards the training samples, impeding the ensemble model, F^t, from generalizing well.

To mitigate Prediction Shift, CatBoost introduces Ordered Boosting [31]. This technique involves selecting a random permutation, π, of the training samples. The learners, $h_1, h_2, ..., h_n$, are then constructed sequentially, ensuring that the learner h_i is trained exclusively on the initial i samples in the training set. CatBoost refines this concept, adapting it to enhance the model's performance further.

Additionally, CatBoost boasts of an exceptionally efficient implementation for both CPU and GPU, offering faster processing speeds than many other gradient boosting frameworks, particularly when managing sizable datasets.

Thanks to these innovations, CatBoost is extensively utilized in both industry and

academia for a variety of machine learning tasks, especially when the priority is predictive accuracy, computational speed, and proficient handling of categorical data.

The Catboost model offer four main advantages which makes them a suitable choice for our machine learning model:

(1) Catboost model provide robust results with relatively small training datasets, making them a suitable choice for tasks with shorter data gathering times.

(2) Catboost model offer fast inference and they are plausible for low-latency enviroments (*e.g.*, Data Centers).

(3) Catboost model require minimal model optimization and hyperparameter tuning. This is advantageous as model optimization is outside the scope of the network operator's responsibility.

(4) Catboost model are explainable, allowing us to discern the aspects of the host that have the most impact on the generated flow sizes in each workload.

2.3.5 Metrics

In this section, we explore the various metrics employed to evaluate and refine our machine learning models, detailing their significance in measuring model accuracy, predictive performance, and guiding the optimization process for enhanced outcomes. A metric is defined as a standardized quantitative measure that assesses, compares, and tracks the performance or quality of a model, serving as an essential tool in the development, testing, and validation phases of machine learning projects. We examine several model evaluation metrics, includ-

ing R^2, MAPE, MSE, and F1 score. Also "Mean SHAP Value" and "CatBoost Feature Importance" as feature evaluation metrics.

- *Mean Squared Error (*MSE*)*: MSE is a measure of the average squared difference between the predicted and actual values in a set of observations. It quantifies the overall accuracy of a predictive model by providing a single value that reflects the average magnitude of errors, providing insight into the model's overall precision. MSE definition is as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

  Where n is the number of data points, y_i is the actual values, and $\hat{y}_i$is the predicted values. It's important to note that MSE is scale-dependent, meaning its value is influenced by the scale of the dataset, which should be considered when comparing models across different data scales.

- *Mean Absolute Percentage Error (*MAPE*)*: MAPE measures the average percentage difference between the predicted and actual values in a dataset. MAPE is used where it is important to understand the relative magnitude of errors. The lower the MAPE, the better the predictive accuracy of the model. MAPE definition is as follows:

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100\%$$

  Where n is the number of data points, y_i is the actual values, and $\hat{y}_i$ is the predicted values. MAPE is particularly valuable for comparing model performance across various datasets or scales because it provides errors in percentage terms, offering an intuitive way

to understand model accuracy without the need for data scaling. However, a notable drawback of MAPE is its inability to compute when any actual value y_i is zero, due to division by zero, rendering it less applicable for datasets with zero or near-zero occurrences.

- R^2: The coefficient of determination, denoted as R^2, is a statistical measure that assesses the proportion of the variance in the dependent variable that is explained by the independent variables in a regression model. R^2 reflects how well the model accounts for the variability in the target value. Higher R^2 values in our experiments means that the chosen features better explain the variabality in the generated flow sizes. R^2 definition is as follows:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

  Where n is the number of data points, y_i is the actual values, $\hat{y}_i$ is the predicted values , and $\bar{y}$ is the mean of actual values. R^2 offers a straightforward and meaningful way to understand how much of the response variable's variation is explained by the model, facilitating the comparison of different models' explanatory power on the same data.

- F1: The F1 score is a metric used in binary classification to assess the model's performance by considering both precision and recall. It is the harmonic mean of precision and recall. Precision measures the accuracy of positive predictions (1 labels), while recall gauges the model's ability to capture all positive instances. F1 is a suitable metric when there is a class imbalance in the dataset.

$$F1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

Where TP is the true positives count, FP is the false positives count, and FN is the false negatives count. The F1 score is primarily used for binary classification problems. For multi-class classification, extensions like the weighted F1 score or micro/macro-averaging need to be used.

- *CatBoost Feature Importance*: In our study, we utilize the "PredictionValuesChange" feature importance metric from CatBoost [31] to assess the impact of individual features on our model's predictions. This method quantifies how much the prediction changes on average when a feature's value is altered during the model's training process. As one of the feature importance metrics offered by CatBoost, it helps identify the most impactful features in the model's decision-making process.

  To understand the calculation of the "PredictionValuesChange" feature importance, let's start with how a leaf node functions: If a data point meets the split condition (which is based on feature F), it moves to the left subtree; if not, it's directed to the right subtree. We use c1 and c2 to represent the total weight of data points in the left and right leaves, respectively. This weight corresponds to the number of data points in each leaf unless specific weights are assigned to them in the dataset. Additionally, we label v1 and v2 as the values attributed to the left and right subtree, respectively.

$$feature\_importance_F = \sum_{leaf_s} (v_1 - m)^2.c_1 + (v_2 - m)^2.c_2$$

$$\text{Where, } m = \frac{v_1.c_1 + v_2.c_2}{c_1 + c_2}$$

- *Mean SHAP value*: We employ Mean SHAP values [32] as a key metric to evaluate the influence of individual features in our models. SHAP (SHapley Additive exPlanations) values create a robust method for understanding how machine learning models make decisions by giving each feature a specific importance value for each prediction. These values reveal the degree to which each feature affects the model's prediction, either positively or negatively.

  Originating from game theory, SHAP values draw on the concept of Shapley values, which distribute a "fair" contribution value to each feature (or "player") within a model (or "coalition") based on their contribution to the prediction (or "game's outcome"). SHAP values are determined by assessing how the presence or absence of each feature alters the prediction, requiring an examination of all possible feature combinations. Despite being calculation-heavy, this method yields accurate insights into feature importance.

  To gauge the significance of features across the entire model, we calculate the Mean SHAP Values by averaging the SHAP values for all predictions. This approach gives us a comprehensive view of how significantly each feature influences the model's predictions, serving as a global indicator of feature importance.

  Additionally, SHAP values approach the model as a "black box," meaning they don't require knowledge of the internal workings of the model to assess feature importance.

This allows SHAP values to be applied universally across various types of machine learning models, providing a consistent method for interpreting model decisions regardless of the model's complexity or architecture.

Chapter 3

Workloads

While there exist publicly available datasets, such as Facebook [33], Google Borg [34] and Alibaba cluster traces [35], representing different workloads in data centers, these workloads fall short of our purposes. In particular, Facebook traces have a 1:30000 packet sampling ratio, lacking the desired flow analysis granularity. Google's Borg traces only include system resource allocation information and do not contain network information. Finally, Alibaba cluster traces are captured in long intervals and do not provide thorough information about system resource allocation.

In order to be able to design an effective system, we need to capture detailed host-level traces in desired time steps. These traces need to be captured during the execution of each workload separately. Therefore, we decided to collect traces from a selection of target workloads.

In this chapter, we explore the specific workloads utilized to assess our flow prediction method, providing insight into their operational mechanisms and the infrastructure supporting their execution. We detail the functionality of these workloads, explaining their sig-

nificance and the scenarios they replicate to challenge and validate our prediction method. Furthermore, the chapter outlines the infrastructure setup employed to facilitate these workloads.

3.1 Intoduction

To encompass a wide range of scenarios, we aimed for a diverse set of workloads, including those that operate on CPUs and others designed for GPUs. The execution of these workloads demanded a robust infrastructure equipped with reliable networking, and access to GPU for the GPU-based workloads or CPU for CPU-based workloads. One of the major hurdles we encountered was establishing such an infrastructure during a period of significant GPU shortages, with many providers reluctant to allocate GPU-capable machines.

Initially, we began with two in-lab machines equipped with GPUs for preliminary tests. However, this setup was far from ideal as it required halting our daily tasks during workload execution due to the shared use of these machines, which were limited in number and featured less powerful GPUs. Consequently, we transitioned to using the ISAIC [36] compute provider at the University of Alberta, known for offering machines with high-performance GPUs. Despite the advantages, we faced challenges with ISAIC's infrastructure, particularly the non-on-demand VM management and the cumbersome process of altering machine configurations through administrative contact, which significantly slowed our progress.

Ultimately, we opted for AWS [37] EC2 G4dn nodes, which provided access to Nvidia T4 GPUs. This platform allowed for full customization of the machines to suit our research needs and came with an initial credit offer of 900CAD, alleviating some financial concerns.

AWS's broad range of services, including EMR nodes for Hadoop environments and S3 for storage, presented a comprehensive solution for our GPU workloads.

For CPU workloads, we began by configuring a network with four lab machines, equipping them with essential tools and frameworks like PyTorch, TensorFlow, Hadoop, and Spark. We then ran the workload code on this lab setup. Ultimately, we transitioned to using AWS for most of the CPU workloads too, attracted by its on-demand VM provisioning and the EMR service, which offers pre-configured Hadoop and Spark nodes, simplifying the setup process.

The execution process involved running the workload code across all machines, followed by our data collector tool on each machine. Upon completion or once sufficient data was gathered, we ceased the workload and data collection operations and transferred the collected traces to storage for further predictive analysis.

Our study focuses on east-west traffic, which involves data exchanges within the data center, typically between servers. This type of traffic is predominant in data-intensive applications where frequent communication between nodes is necessary. All of our selected workloads, including distributed deep learning training and Spark workloads, generate east-west traffic. This choice was driven by the need to explore the dynamic interactions within a data center environment, where nodes continuously exchange information to perform complex computations.

We attempted to incorporate systems that handle north-south traffic, which involves data transfer between the data center and external networks. However, the generated flows from these workloads were highly predictable, largely due to the challenges in accurately simulating real user behavior. The predictability of north-south traffic did not provide a

suitable testbed for our flow size prediction models, as it did not reflect the complexity and variability found in east-west traffic.

3.2 AWS

For the infrastructure supporting our workloads, we utilized AWS (Amazon Web Services). AWS is a comprehensive and widely adopted cloud platform that offers over 200 fully featured services from data centers globally. It provides a variety of computing power, storage, and other cloud services, enabling users to deploy and manage applications and services through the internet. AWS's scalable and flexible services support a wide range of applications, making it an ideal choice for hosting our deep learning and spark workloads.

For all our Spark workloads, we utilized an Amazon EMR cluster composed of four nodes: one task node, one core node, and one primary node, each configured as an m5.xlarge machine. Amazon EMR (Elastic MapReduce) is a cloud service designed to process vast amounts of data efficiently, using popular big data frameworks like Apache Spark and Hadoop. In this setup, the primary node oversees task distribution and overall cluster management, core nodes handle data storage and task execution, and the task node provides additional processing power for executing tasks. The m5.xlarge machine type is known for its balance of compute, memory, and networking resources, making it well-suited for a wide range of workloads, including our Spark tasks, ensuring efficient processing and analysis of large datasets within our distributed computing environment.

For our GPU-based deep learning training, we opted for 3 G4dn instances on Amazon EC2, which come equipped with NVIDIA T4 GPUs, tailored for high-performance machine

learning tasks. Meanwhile, for CPU-based deep learning training environments, we utilized 3 c5.xlarge machines. Amazon EC2 (Elastic Compute Cloud) provides resizable compute capacity in the cloud, allowing users to scale up or down based on their computational requirements. It offers a wide variety of instance types designed to fit different use cases, from intensive compute tasks to data processing. This flexibility made EC2 an ideal choice for our varied deep learning workloads, providing the necessary computational power whether we needed the advanced processing capabilities of GPUs or the robust performance of CPUs.

We extensively utilized Amazon S3 [38] to manage our datasets and trace files. Amazon S3, or Simple Storage Service, is a scalable object storage service provided by Amazon Web Services (AWS). This service enables us to store and retrieve any amount of data at any time from all of our nodes on all of the workloads, making it ideal for storing large volumes of data in our research.

3.3 Deep Learning Workloads

Deep learning [39] is a subset of machine learning that constructs and utilizes neural networks with several layers, or "deep" networks, to process vast quantities of data. These multilayered structures enable the model to learn and interpret complex patterns and features at various levels of abstraction. Deep learning techniques have revolutionized fields such as image recognition [39], natural language processing [40, 41], and pattern identification, achieving remarkable accuracy. By leveraging extensive training datasets and significant computational power, deep learning models can perform a wide range of tasks with precision that often surpasses traditional machine learning approaches.

Training a deep learning model requires processing vast datasets through a neural network and tuning the network's settings to improve prediction accuracy. Through a method called backpropagation, the model systematically updates the strengths of the connections between the network's nodes to reduce discrepancies between its predictions and the real outcomes. The objective is to enhance the model's parameters until it can reliably make accurate predictions on new, unseen data.

In this part of the chapter, we explore the deep learning workloads we employed. Our analysis includes two separate distributed deep learning training setups, each utilizing different hardware on the AWS EC2 platform. The first setup is training of a small GPT Model which is implemented using PyTorch, and runs on a three-node GPU cluster. Each node in the GPU cluster is powered by an NVIDIA T4 [42] GPU. The second setup is training of an image recognition model based on CNNs which executed on a three-node CPU cluster and implemented using TensorFlow.

Both setups are designed to operate with distributed data, a method that efficiently manages large datasets by dividing them across several nodes in the cluster. This technique boosts computational efficiency and cuts down on the time needed to train intricate deep learning models. We will delve deeper into the details of distributed training and the processes that support it in 3.3.1.

3.3.1 Distributed Deep learning training

There are two main strategies for distributing the deep learning model training on multiple machines: distributing the data or the model. It's also possible to use both approaches at

the same time. In the **Data-Parallel** method, the training dataset is split into smaller parts, and these parts are spread across various computing units. Each unit works on its portion of the data, calculating adjustments needed for the model's parameters. Once processed, these adjustments (like gradients for neural networks) are combined to update the entire model. This technique of processing in parallel speeds up the training process for large datasets by using the collective power of several machines or processors. In the **Model-Parallel** approach, each worker node processes identical copies of the complete dataset but focuses on different segments of the model, making the model a collective of its segmented parts. The model can be divided into various segments, such as individual layers, groups of layers, or specific components within a layer, based on the model's structure and available computing power. Each segment is then allocated to a different processing unit for parallel processing. For example, one GPU may compute the first several layers, while another takes on later layers. Synchronization is crucial in model parallelism to ensure accurate execution of both the forward pass and the backward pass, including gradient computation and backpropagation during training. Processing units must coordinate to receive and send necessary data between segments for continuous computation [43].

In distributed deep learning, the structure of communication, known as topology, significantly impacts how data, computations, and model parameters are exchanged and coordinated across the network of machines participating in the training. The choice of topology is key to the distributed training's effectiveness, capacity to scale, and overall performance. There are three widely used topologies: Trees, Rings, and Parameter Servers, each with unique characteristics that influence the training process. Since our deep learning training workload use Ring topology, we explain this strategy in more depth.

In a ring topology, every machine or device is linked to two others, creating a circular layout. Data, gradients, or parameters are passed around this loop, with each node getting data from one neighbor, processing it, and then forwarding it to the next. This setup is effective for moving data as it evenly distributes the workload across all nodes and each node only needs two connections, reducing bottleneck risks. Nonetheless, the time it takes to finish a task depends on the ring's size since data has to go through each node one by one.

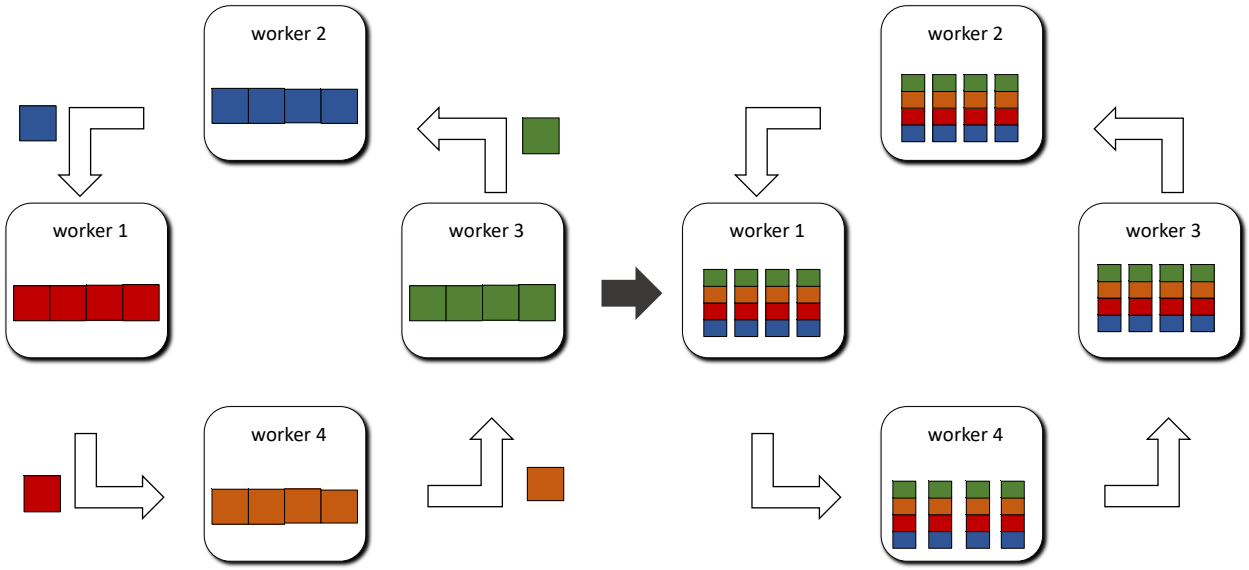


Figure 3.1: The ring architecture for distributed training. The data distributed among all nodes and the same model replicated on all of the nodes. In each iteration the gradient of that iteration will be sent to the next node.

Every topology comes with its own set of benefits and drawbacks, and selecting one over the others depends on the particular needs of the distributed deep learning project, such as the model's size, the volume of data, and the computing resources at hand.

In distributed deep learning, the selection of a communication primitive is a key consideration. These primitives are vital for sharing information, like model parameters and gradients, across a network's nodes. Primitives such as all-reduce, broadcast, gather, and

scatter play a pivotal role in ensuring the coordinated training of deep learning models on multiple machines. Frameworks such as MPI (Message Passing Interface), NCCL (NVIDIA Collective Communications Library), Horovod, and Gloo efficiently implement these operations, facilitating smooth data parallelism and model updates. The efficiency of these primitives greatly affects the speed and scalability of training deep learning models, making it possible to work with large models and datasets over diverse computational setups.

3.3.2 Pytorch minGPT Model Training Workload

PyTorch [44] is a widely-used open-source framework for machine learning, known for its flexibility and user-friendly nature. It offers a rich set of tools and libraries for deep learning applications, such as image and language processing. PyTorch supports computations on both CPUs and GPUs, which accelerates the development and training of complex models with large volumes of data. One of its key features is the ability to perform distributed training, allowing for Data parallel and Model parallel strategies, thus shortening training periods and efficiently managing vast datasets.

We used a simplified version of the GPT (Generative Pretrained Transformer) [45] model, named minGPT, as one of our workloads. GPT is a neural network architecture that uses the mechanism of transformers to generate text by predicting subsequent words in a sequence given the words that precede them. The training of GPT models allows them to understand and generate language with coherence and context relevance. The minGPT is a more compact version of this model, designed for efficiency and ease of use in smaller projects or for educational purposes. It retains the essential features of the original GPT but is opti-

mized for less resource-intensive applications, making it suitable for our multi-GPU training experiments. In minGPT, there are total of 27.32 Million parameters.

We trained the model using the distributed data parallel strategy of PyTorch implemented with NCCL as communication primitives framework. The core of this strategy is implemented in `DistributedDataParallel` class in PyTorch. This class uses the Ring topology for communication among the training nodes. The code for the training of minGPT is provided on github[1].

For our study, we trained the GPT model on a dataset comprised of Shakespeare's works. This dataset, often referred to as the Shakespeare dataset, includes a wide range of the playwright's compositions, such as sonnets, plays, and poems. It is commonly used in natural language processing (NLP) tasks to train models on complex, stylistic text, providing a rich source of Elizabethan English vocabulary and syntax. Training the GPT model on this dataset creates a model with ability to understand and generate text that mimics Shakespeare's unique literary style.

To facilitate the management of distributed training processes across nodes for our PyTorch workloads, we utilized torchrun [46]. This utility offers robust process management by efficiently handling process failures, including the graceful restart of any failed processes. Additionally, torchrun supports dynamic scaling, allowing for the adjustment of the number of nodes involved in the training process, either by scaling up to include more nodes or scaling down as necessary.

For the collection of traces from the PyTorch distributed training workload, we initiated by setting up three G4dn nodes within the AWS EC2 environment. Subsequently, our

[1] `https://github.com/mory91/distributed-dl/tree/main/mingpt-pytorch`

Trace Collector was deployed on each node, configured to commence trace collection and store these in a designated directory on their respective hard drives. The training process was then initiated individually on each node using torchrun. After an hour of running the training, we terminated the training process and transferred the accumulated traces to a designated bucket on AWS S3 for further analysis.

3.3.3 Tensorflow Image Recognition Workload

For our second deep learning task, we employed TensorFlow [47] to train a model across four CPU nodes. TensorFlow is a open-source software library for numerical computation, particularly well-suited for large-scale machine learning and deep learning tasks. It allows developers to create complex models with ease, thanks to its flexible architecture and extensive library of tools. In this case, we utilized TensorFlow to train a straightforward Convolutional Neural Network (CNN) model for image recognition. The CNN Model, simply consists of a single CNN layer followed by a fully connected layer with ReLU activation, and a linear classification head at the end of the network. The training was conducted on the MNIST dataset, a standard benchmark in the field that consists of handwritten digits. The CNN Model was trained to recognize the hand written digits in the MNIST dataset. The CNN model has roughly 300000 parameters.

We used the data parallel strategy implemented in Tensorflow for multi node CPU-based environments. The core of the data parallel strategy of Tensorflow is implemented in `MultiWorkerMirroredStrategy` class. In this strategy, Tensorflow uses ring architecure for communication among the nodes. In a CPU-based enviroment, the Tensorflow uses it's own

default implementation of communication primitives (*e.g.*,`all-reduce`, `broadcast`, `gather`). The code for the training of the workload is provided on github[2].

To train the CNN model, we set up four CPU-based nodes on AWS. We install the Trace Collector on each node. Then, we initiated the training scripts sequentially across the nodes. Shortly after starting, the Tensorflow training processes synchronized, and the training starts. After one hour of training and trace collection, we terminated the training and moved the gathered traces to a dedicated S3 bucket on AWS for further analysis.

3.4 Spark and Hadoop

In this section, first we start with a brief introduction on hadoop [48] and spark [49], then we delve into the specific workloads executed on Hadoop Spark, leveraging the AWS Elastic MapReduce (EMR) service [50]. This section is structured into subsections, each dedicated to a distinct Spark workload that was utilized in our research: Spark KMeans, Spark PageRank, and Spark SVM. AWS EMR [50], renowned for its managed Hadoop framework, provides an optimized environment for big data processing and analysis, enabling us to efficiently run these advanced Spark workloads. Each subsection will provide an overview of the workload, including its implementation details.

Hadoop

Hadoop [48] is an open-source framework designed for the distributed storage and processing of large data sets across clusters of computers using simple programming models. Developed

[2] `https://github.com/mory91/tf-dist`

by the Apache Software Foundation, Hadoop is aimed at providing a scalable, efficient, and fault-tolerant solution for big data analytics.

Hadoop's architecture comprises three key components:

1. **Hadoop Distributed File System (HDFS).** HDFS [51] forms the storage layer of Hadoop, engineered to store massive data sets reliably across multiple machines within a large cluster. It ensures reliability through the replication of data blocks across various nodes, enabling the system to withstand failures without losing data.

2. **Yet Another Resource Negotiator (YARN).** YARN [52] acts as the resource management layer of Hadoop, tasked with managing and allocating resources throughout the cluster. It facilitates the efficient handling of data stored in HDFS by various data processing engines.

3. **MapReduce.** MapReduce [53] is a programming model tailored for processing large data sets using a parallel and distributed algorithm across a cluster. It operates in two phases: the *Map* phase, which filters and sorts the data, and the *Reduce* phase, which aggregates the data into a summary form.

Hadoop operates on a master-slave setup, with the master managing, scheduling, and monitoring tasks performed by the slaves. In HDFS the master node is named the NameNode, which handles metadata, while the slave nodes, known as DataNodes, are responsible for storing the real data. Likewise, within YARN the ResourceManager serves as the master, coordinating resource allocation, and the NodeManager, located on each slave node, executes the tasks across the network.

Hadoop clusters are scalable, meaning they can grow by adding more nodes, which increases their capacity to store and handle larger amounts of data. The system operates on affordable, commodity hardware, making it an economical choice for managing and analyzing substantial data volumes. Furthermore, Hadoop ensures data safety through automatic replication across multiple nodes, safeguarding against data loss during hardware malfunctions and guaranteeing high availability. Additionally, Hadoop facilitates machine learning projects by supplying ample data necessary for training algorithms.

Spark

Apache Spark [49] is an open-source, distributed computing system that offers an interface for programming entire clusters with implicit data parallelism and fault tolerance. Spark has gained popularity in big data and machine learning for its fast processing, efficiency, and user-friendly design. It's designed to perform both batch processing (similar to MapReduce) and new workloads like streaming, interactive queries, and machine learning.

Spark enhances the performance of applications on Hadoop clusters, allowing them to run up to 100 times faster in memory and 10 times faster on disk. This increase in speed comes from its advanced DAG (Directed Acyclic Graph) execution engine that streamlines the execution of tasks. Additionally, Spark provides user-friendly APIs for large dataset operations and a suite of higher-level tools such as Spark SQL for managing SQL and structured data, MLlib [54] for machine learning, GraphX [55] for graph analysis, and Spark Streaming for live data processing. Its DAG execution engine also ensures fault tolerance, automatically rerunning tasks if a part of the execution fails, thus maintaining the system's reliability.

Spark and Hadoop are both pivotal in the big data ecosystem, serving complementary functions. Hadoop is renowned for its Hadoop Distributed File System (HDFS) which offers exceptional storage capabilities, while Spark is designed for rapid data processing. Spark can operate atop HDFS, harnessing Hadoop's reliable storage and scalability to manage data, coupled with Spark's processing speed. Spark is also able to work with various storage systems, such as Amazon S3 or Apache Cassandra, although its integration with Hadoop's ecosystem is notably smooth, making it ideal for projects already embedded in Hadoop. Additionally, while Spark can function independently, it frequently runs on clusters managed by Hadoop's YARN (Yet Another Resource Negotiator), ensuring efficient resource distribution and task scheduling across applications. Together, Hadoop's storage prowess and Spark's analytical speed deliver a holistic solution to big data's storage and processing needs, encapsulating the essentials for handling and analyzing vast datasets effectively. In Figure 3.3 the high-level architecture of spark running on hadoop is illustrated.

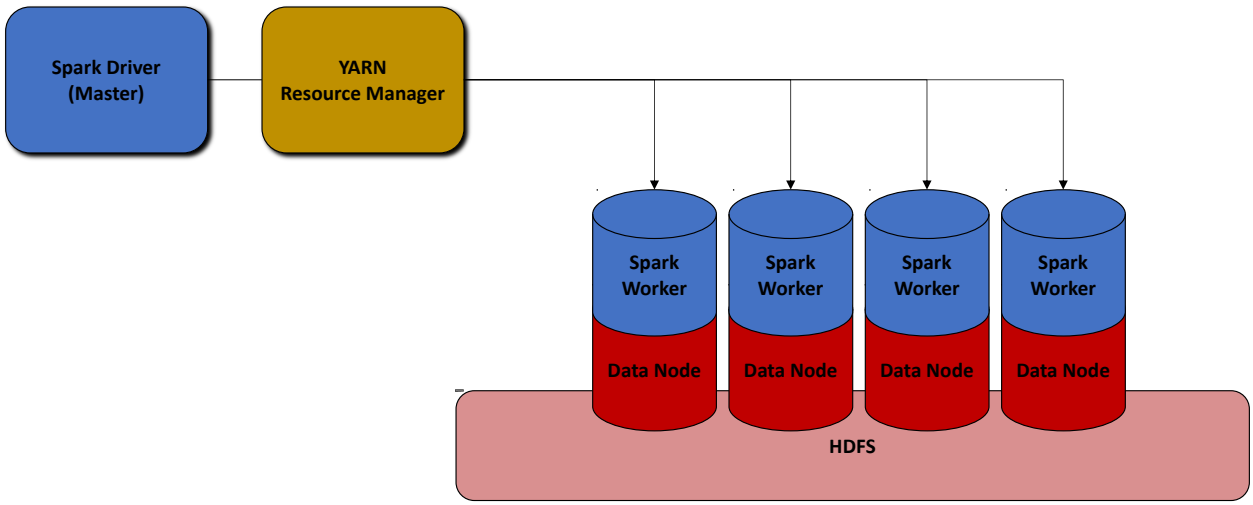


Figure 3.2: Spark and Hadoop overall architecture.

3.4.1 Spark Kmeans Workload

Apache Spark's MLlib [54] is a library offering machine learning algorithms, such as KMeans, built on Spark's platform. KMeans is a widely used clustering technique that divides a dataset into distinct, separate clusters. Within Spark MLlib, KMeans is optimized for processing large datasets by utilizing Spark's ability to distribute computing tasks across multiple nodes, thereby enhancing the speed of the clustering operation.

Due to the potential for suboptimal clustering outcomes and delayed convergence caused by the random selection of initial centroids in the traditional KMeans algorithm, Spark's MLlib employs a enhanced parallel version called KMeans—— [56]. This variant is designed to refine the algorithm's initialization phase, making it more effective for use in distributed computing settings such as Spark.

KMeans—— starts by randomly selecting one initial centroid from the dataset. It measures the distance from every data point to this centroid. Then, it picks new centroids based on a probability related to the square of their distance to the nearest existing centroid, repeating this process through several iterations. This approach generates a varied set of initial centroids that are well-distributed over the data space. From this broader pool of potential centroids, a weighted sample is chosen as the initial centroids for the regular KMeans algorithm, with the selection weighted by distance to ensure a wide distribution. These initial points set the stage for the traditional KMeans process, where data points are assigned to the closest centroid, and centroids are then updated accordingly.

We use Spark's MLlib KMeans for our workload, showcasing its use in iterative distributed machine learning, where each iteration involves minimal data communication. Our chosen

dataset for Kmeans clustering is MNIST [57]. The MNIST dataset is a large collection of handwritten digits, commonly used for training various image processing systems. It contains $70,000$ images of handwritten digits (0 through 9), divided into a training set of 60,000 images and a test set of 10,000 images. Each image is a 28x28 pixel grayscale representation of a digit.

3.4.2 Spark Pagerank Workload

We included PageRank, as implemented on Spark, as one of our workloads. PageRank [58], originated by Google, evaluates the significance of web pages through the analysis of their links' quality and structure. While its initial purpose was to enhance search engine rankings, PageRank's applications have expanded into areas like social network analysis and recommendation systems, showcasing its versatility beyond just web page ranking.

PageRank is based on the idea that important web pages are usually linked to by many other sites. However, not every link has equal importance; a link from a well-regarded site is more valuable than one from a lesser-known site. The algorithm gives each web page a numerical score to indicate its importance in a network of linked documents, like the World Wide Web. This score helps determine the page's relative significance within the network.

PageRank initially assigns the same PageRank value to every page in the dataset. It then recalculates each page's PageRank by considering the PageRank of pages linking to it. This process repeats across several iterations, with each page's PageRank updated by accumulating a share of the PageRank from its linking pages. To mimic a web user's random browsing habits, PageRank uses a damping factor, typically around 0.85, which reflects the

likelihood of continuing to click links versus starting over or stopping. This factor helps the algorithm reach a stable set of PageRank values. After multiple iterations, PageRank values are normalized to ensure their sum equals one, thereby creating a probability distribution.

We executed the PageRank algorithm on the Twitch gamers dataset [59] as part of our analysis. The Twitch gamers dataset comprises data from the Twitch streaming platform, focusing on users, commonly known as gamers, and their interactions. It includes information about which games are being played, the connections between different gamers through follows or subscriptions, and the social network structure within the Twitch community. This dataset is valuable for understanding the popularity and influence of gamers within the Twitch ecosystem, making it an ideal candidate for analysis with the PageRank algorithm to identify key influencers and relationships.

3.4.3 Spark SVM Workload

In our study, we incorporated the training of a Support Vector Machine [60] (SVM) model using Stochastic Gradient Descent (SGD) on the Spark platform as one of our key workloads. SVM is a cornerstone of supervised learning, highly valued for its precision in classification tasks. This algorithm works by finding the best hyperplane to divide different classes within the dataset. When dealing with data that cannot be separated linearly, SVM uses kernel functions to transform the data into a space where linear division is possible. The integration of SGD [61], an optimization technique that minimizes the objective function iteratively, enhances SVM's applicability to large-scale datasets. Unlike the conventional gradient descent that relies on the entire dataset for each update, SGD updates model parameters using only

a small subset of the data at a time, significantly improving computational efficiency.

The MLlib library in Apache Spark simplifies the process of executing SVM with SGD across a distributed computing setup. By utilizing Spark's powerful distributed architecture, the algorithm can efficiently handle data that is distributed among various nodes in a cluster. This approach effectively tackles the issue of managing large datasets that surpass the memory limits of individual computers.

For our dataset, we selected the Display Advertising Challenge dataset provided by Criteo Labs [62]. This dataset is a collection of data that Criteo Labs released for the purpose of developing models capable of predicting ad click-through rates (CTR). It includes millions of records, each representing a user interaction with an advertisement. The dataset features a mix of anonymized categorical and numerical variables that describe various aspects of the ads, user demographic information, and context of the interaction. The goal with this dataset is to use the provided features to accurately predict whether or not an ad will be clicked, which is a common challenge in online advertising optimization.

3.4.4 AWS EMR Setup

We utilized AWS EMR [50] as the platform for our Hadoop cluster, which is comprised of three nodes: a data node, a task node, and a primary node. The primary node manages the cluster and coordinates the distribution and processing of data. The data node is responsible for storing large volumes of data and executing tasks as directed by the primary node. Lastly, the task node is specifically allocated for processing tasks, handling the computational operations needed to analyze the data stored across the cluster. In our cluster configuration,

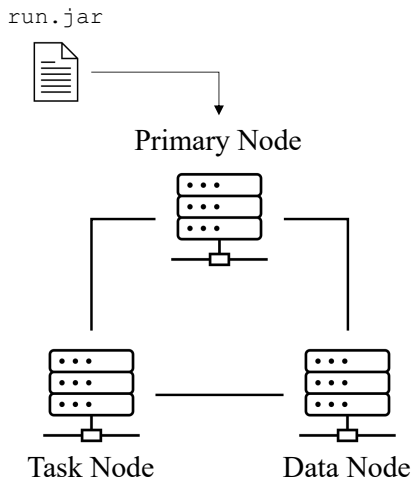


Figure 3.3: The architecture of our AWS EMR cluster. The cluster consists of 3 nodes, a data node, a task node and a primary node. The workload compiled jar file (*e.g.*,`run.jar`) is deployed on the primary node using *spark-submit* tool.

each node is an m5.xlarge type, equipped with 8 vCore CPUs and 16 GiB of memory.

To execute each workload on the cluster, we developed the necessary code in the Scala language. The datasets required for each workload were uploaded to AWS S3 to ensure they were accessible from all of the nodes in the cluster. We then compiled the Scala code into a JAR file and deployed it to the primary node of the cluster. Alongside, our Trace Collector was installed on each node to begin trace collection.

Workloads were initiated from the primary node using the *spark-submit* tool, which is responsible for distributing and managing the application's execution across the cluster. This tool efficiently manages the task distribution and execution within Spark, ensuring optimal resource use and workload balancing across the cluster. The source code for the spark workloads is provided on github[3].

[3]`https://github.com/mory91/spark-examples.git`

Chapter 4

Design and Implementation

4.1 Introduction

In this chapter we describe trace collection process and trace collector, the machine learning method and the training procedure, and the details of the features extracted for the model.

The system traces that we use as features includes a wide range of different host-level metrics. These metrics vary from usage amount of different memory segments, data read from (or written to) disk, to memory allocations executed by the application code using specific libraries (malloc, jmalloc, etc.).

4.2 Trace Collector and Collection Process

The trace collector is responsible for capturing traces within each host of the workload. It records three main types of traces from each host: 1. Packet Trace, 2. Polling Traces, and 3. Event Traces. The trace collector runs as a background process independently inside each

host of the workload. In this section we detail the trace collection of each trace type.

4.2.1 Packet Trace

The trace collector captures the packets from the hosts by using a packet capture sub-module implemented inside the Trace Collector. The packet capture sub-module collects the essential packet data, including IP and port for the source and destination of the packets, as well as their size and arrival time, from each packet involved in communication between hosts of the target workload. Since the packet transfer rate can be very high depending on the workload, this sub-module needs to work at high performance to capture all packets.

We utilize the *PF_RING* library [22], which provides a high-speed packet capture mechanism, to manage high packet transfer rates. To conserve memory, we store only essential packet header information (*e.g.*, size, arrival time, addresses, *etc.*) instead of the entire packet data. A detailed description of *PF_RING* library is provided in section 2.3.3.

4.2.2 Polling Traces

The Polling Traces are high-level traces which provide the statistics of a given process from the operating system point of view. Polling traces are captured using monitoring tools provided by the operating system. The focus of our work is the Linux-based operating systems. The *"/proc"* directory in the Linux OS includes a variety of files and directories that provide a broad collection of resource statistics. Each file or directory located within the *"/proc"* directory fulfills a specific role and provides information regarding various aspects of the system's utilization of resources. For example the virtual memory usage of a target

workload with a specific *pid* is stored in a column in the *"/proc/[pid]/stat"* file. The Trace Collector periodically polls necessary statistics from specific files in the *"/proc"* directory to create each trace. To avoid accuracy degredation, the polling interval should be shorter than the flow time gap, capturing concise information for each flow.

The Trace collector captures multiple traces of the Polling Trace type. Each polling trace, is a trace of a specified system information captured at each timestep during the workload's execution (*e.g.*, The trace of memory usage is the memory usage of process recorded from *"/proc/[pid]/stat"*).

4.2.3 Event Traces

These traces correspond to function call events that occur during the execution of the workload. These functions can take the form of a system call or a userspace-defined function call. The Trace Collector captures this trace type using eBPF technology in Linux. This technology allows the definition of hooks for various events that execute when the event occurs. A detailed description of eBPF is presented in section 2.3.1.

Events can encompass different function types, such as the invocation of a *read* system call by a program, a library defined function such as *malloc*, *etc.*. We define hooks for the desired events to extract necessary information (*e.g.*, return value of the function, time of function execution) from the event. The Trace Collector records multiple event traces corresponding the execution of various functions. For instance one of these traces is associated with the *tcp_sendmsg* system call, and the Trace Collector records the return value (*i.e.*, the passed buffer size) and the time of the *tcp_sendmsg* function execution. It is important to notice in

different types of workloads, different set of events are tracked. For example for deep learning training on GPU workloads, we trace the CUDA library function calls (*i.e.*, `cudaMalloc`, `ncclAllReduce`) as our events.

In order to utilize eBPF, our approach incorporates the BCC (BPF Compiler Collection) framework. BCC provides essential tools and libraries for creating efficient kernel tracing and manipulation programs using eBPF. For each trace, we developed a BCC function that captures the necessary data from system events. Our Trace Collector employs these BCC functions, initiating them to capture pertinent data and ceasing their operation once trace collection process is finished.

Each event tracer which is written in BCC framework is divided into two components: a Python script running in user space and a segment of C code operating in kernel space. The Python script's primary function is to load the eBPF code into the kernel, utilizing the `attach_*` functions to hook the eBPF code to specific system events. Additionally, it handles the retrieval of emitted events from the kernel by implementing a callback mechanism that captures event data and records it to an output buffer. On the other hand, the C code in kernel space is triggered upon the invocation of the targeted function. It generates an event object during each function call and transmits this data back to user space through the eBPF's ring buffer, a high-performance data structure designed to pass information between the kernel and user space efficiently, without needing system calls for each event.

For instance, the user space code responsible for monitoring the `tcp_sendmsg` system call is detailed in 4.1. We employ the `attach_kretprobe` function to link our kernel space function to the return of the `tcp_sendmsg` system call. The kretprobe (kernel return probe) is a dynamic tracing tool that allows us to intercept the return of kernel functions; it is utilized

```python
import sys
import argparse
from send import get_bpf, get_call_back
# argument parsing
parser = argparse.ArgumentParser()
parser.add_argument("-p", "--pid", default=None)
args = parser.parse_args()
if args.pid is None:
    print("PID must set")
    exit(0)
# setting up ebpf
bpf_obj = get_bpf(args.pid)
bpf_obj.attach_kretprobe(event="tcp_sendmsg", fn_name="send_return")
cb = get_call_back(bpf_obj)
bpf_obj['events'].open_ring_buffer(cb)
# reading events
while True:
    try:
        bpf_obj.ring_buffer_consume()
    except KeyboardInterrupt:
        exit()
    sys.stdout.flush()
```

Listing 4.1: User Space code for `tcp_sendmsg` function tracing

here to capture the post-execution context of the `tcp_sendmsg` call, enabling analysis of the call's outcomes. Once the kernel space function is loaded, we begin the process of extracting data from the ring buffer, continuously pulling event data that has been recorded by the eBPF program as system calls are executed and completed.

The kernel space function utilized for tracing the `tcp_sendmsg` system call is detailed in 4.2. This function leverages kernel-defined functions like `bpf_ktime_get_ns` and `PT_REGS_RC` to extract data from the execution context. Specifically, we capture the size of the buffer passed to the `tcp_sendmsg` and the timestamp of the execution. This information is then dispatched to the user space through the ring buffer, where it can be processed further.

In Figure 4.1 the overall architecture of The Trace Collector is shown.

```
#include <uapi/linux/ptrace.h>
#include <net/sock.h>
#include <bcc/proto.h>

struct event {
        int pad;
        int size;
        u64 timestamp_ns;
};
BPF_RINGBUF_OUTPUT(events, 1 << 12);

bool send_return(struct pt_regs *ctx)
{
    u64 id = bpf_get_current_pid_tgid();
    if (id >> 32 != __PID__) { return 0; }
    int size = PT_REGS_RC(ctx);
    u64 time = bpf_ktime_get_ns();
    struct event event = {
        .size = size,
        .timestamp_ns = time
    };
    events.ringbuf_output(&event, sizeof(event), 0);
    return true;
}
```

Listing 4.2: Kernel code for `tcp_sendmsg` function tracing

Since we need to have data that is only related to the runtime and communication of the nodes involved in our target workload, we deploy the Trace Collector on each host while the target workload is running on the hosts.

The data collected from the trace collection process is stored in multiple text files. One of the files contains the workload packet trace, with each line detailing necessary packet data including arrival time, size, and address, among other metrics. The remaining files are dedicated to polling traces and event traces, with each file corresponding to a specific type of trace. In these files, each line records the time of the event or polled value, along with the associated value, providing a structured format for analyzing the behavior and interactions

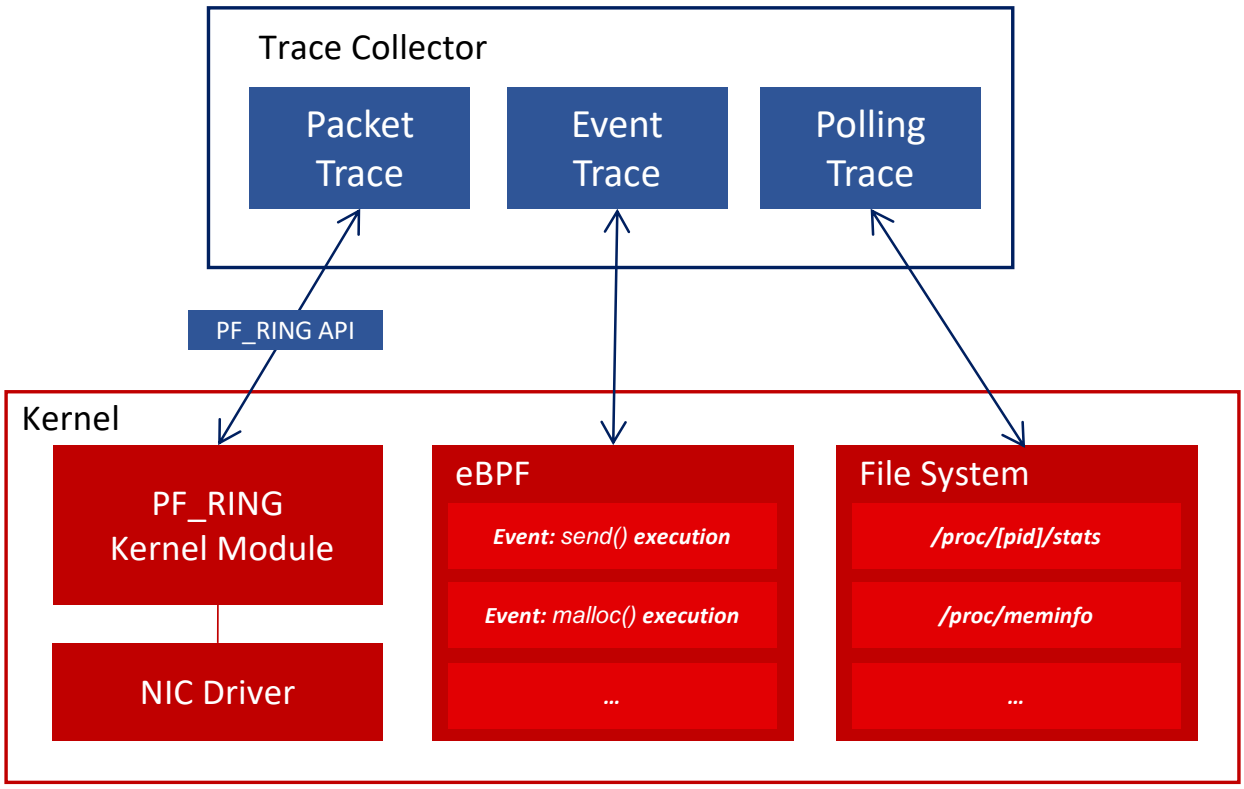


Figure 4.1: The overall trace collector architecture.

within the process.

The trace collector is written in Go and it is open-sourced in github[1].

The reason behind collecting all these traces is to gain a comprehensive understanding of the interactions within the system and between the system and the network. By capturing a wide range of trace types—packet traces, polled system traces, and event-based traces—we can build a detailed view of the system's behavior and the factors influencing network flow sizes. The diagram in Figure 4.2 shows a bird's eye view of the system and its resources, illustrating how different traces are collected.

For example, when the application issues an `ncclAllReduce` call, it results in a network-related system call (e.g., `sys_send`), which in turn creates packets that we capture. By collecting traces at multiple levels, such as the application level (`ncclAllReduce`), the system

[1] `https://github.com/mory91/ogomon.git`

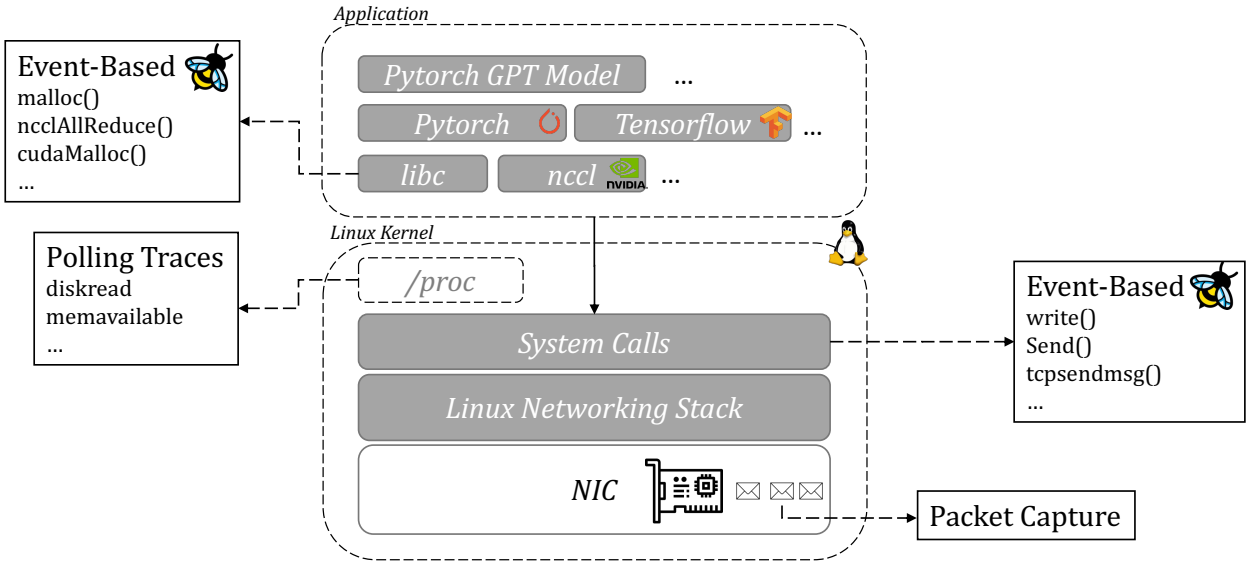


Figure 4.2: Overview of the system showing the trace collection process. The diagram illustrates how various traces are captured at different levels, including event-based traces (e.g., `malloc()`, `ncclAllReduce()`, `cudaMalloc()`), polled system traces (e.g., diskread, memavailable), and packet captures. These traces are collected from the application level, the Linux kernel level, and the networking stack, providing a detailed view of the system and network interactions. This comprehensive data collection helps in accurately modeling and predicting network flow sizes..

call level (`sys_send`), and the packet level, we obtain a comprehensive dataset that allows us to accurately model and predict network flow sizes. This holistic approach ensures that we do not miss any critical interactions that could affect the accuracy of our predictions.

By understanding the detailed interactions between various components of the system and network, our models can more accurately predict flow sizes, ultimately leading to better network management and performance optimization.

4.3 Dataset Construction

After the data is gathered for the training phase of the model, we run a pre-processing module on the data to extract the flows using the time of each record from the raw system traces and packet captures.

4.3.1 Flow Dataset

After the trace collection, we need to construct the dataset for our model. For a target workload, we collect the traces once, and create distinct datasets from the traces for each flow time gap. This means that since we examine 35 distinct flow time gaps (*i.e.*, from 500 to 20000 microseconds with 500 micorseconds steps) we create 35 different datasets. The dataset which is extracted from the traces has a tabular format where each row corresponds to a flow, and each column represents an individual feature associated with that flow. To construct the dataset for each flow time gap, we go through the following procedures:

1. **Flow Extraction.** We extract the flows from the packet trace collected by the trace collector. Each packet in our packet trace has the source IP, source port, destination IP, destination port, packet size and arrival time for all all of the packets communicated from and to the host during the workload execution. Since multiple hosts participate in the workload execution, in the packet trace there are in-bound and out-bound packets with different IP adrresses and different ports. To extract flows, we group the packets by their source IP, destination IP, source port and destination port, to get the group of packets on each individual comminication path (*i.e.*, Path from *ipA:portA* to *ipB:portB*) from the specified host. To extract flows from each group, we find the set of packets which has a time gap more than the specified flow time gap to the next packets set. In this way, multiple flows are extracted from each path with distinct set of packets. The same flow extraction procedure is done for each flow time gap. In Figure 4.3 we show the process of flow extraction on a sample path.

2. **Feature Extraction.** After extracting the flows, we proceed to extract features for each

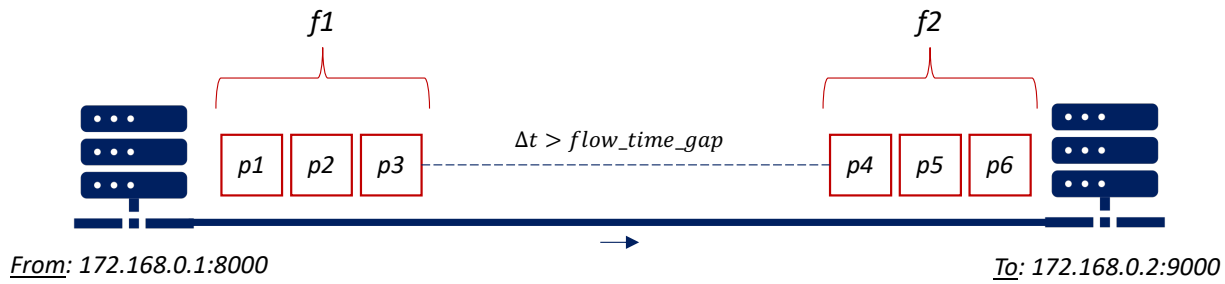


Figure 4.3: Two flows (f1 and f2) extracted from the path beween *172.168.0.1:8000* to *172.168.0.2:9000*. Flow 1 has packets p1, p2, p3 and flow 2 has packets p4, p5, p6. The gap time between the two sets of packets is greater than the flow time gap Δt.

flow. Each feature used by the model is extracted from one of the three different types of collected traces which trace collector records from the host. Each trace type undergoes a distinct feature extraction procedure.

To extract network-related traces, we aggregate packets for each individual flow from the packet trace, from which all network-related features can be derived. For example *network-out* feature, is the sum of the sizes of all of the packets arrived from start of the workload execution until the start of the flow.

In order to extract features from the polled traces, we get the last polled trace value before the start of the flow and consider the value for the feature. For example for the trace of the *rss_memory* we locate the last recorded value in the *rss_memory* trace, and use that as the *rss_memory* feature value of the flow. For event-based traces, we extract two features from each trace. The first one is the value of the last event happened before the start of the flow, and the second one is the amount of time between that event and the start of the flow. We name those features with the format of *feature_name* and *tt_feature_name* respectively. In Figure 4.4 we can see the procedure of the feature extraction from event-

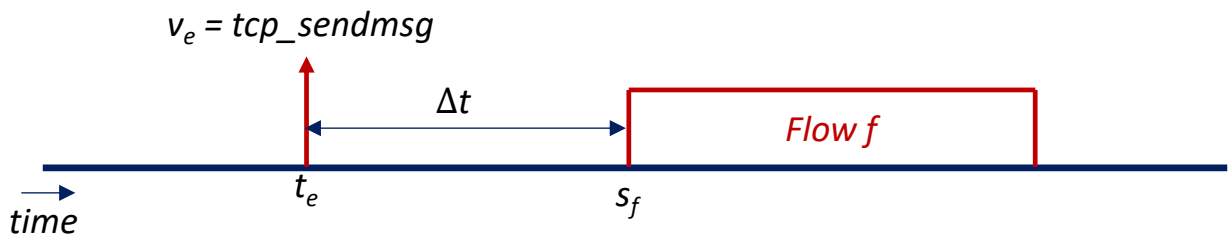


Figure 4.4: To extrace the *tcp_sendmsg* feature for the flow f with the start time s_f, we find the last call of *tcp_sendmsg* (e) before s_f. The return value v_e will be the *tcp_sendmsg* feature value and $s_f - t_e$ will be the *tt_tcp_sendmsg* feature value.

based traces.

We assess the model using two feature sets: the *"limited feature set,"* which contains fewer features, and the *"extended feature set,"* which comprises an extensive set of features. In Table 4.1, the list of all features employed by the system is provided, with feature names marked with (*) denoting the subset exclusive to the *"extended feature set."*

3. **Historical Information.** To incorporate historical information about past flows into our model, we include the features of K previous flows for each current flow. Also, we add previous flow sizes as historical information features. The features of the t-th flow before the current one are added under the name *feature_name**t***. Through experiments, we have determined that the optimal value for K in our workloads is 5. Using fewer than 5 previous flows resulted in underfitting, while including more than 5 previous flows increased the risk of overfitting.

4. **Labeling.** We approach the flow prediction problem from two distinct methodologies. In the first, we undertake a regression-based approach where our aim is to predict the precise size of the next flow generated by the host. In this setting, the prediction target for each flow, will be the size of the next flow. We name this approach as the *Regression-based*

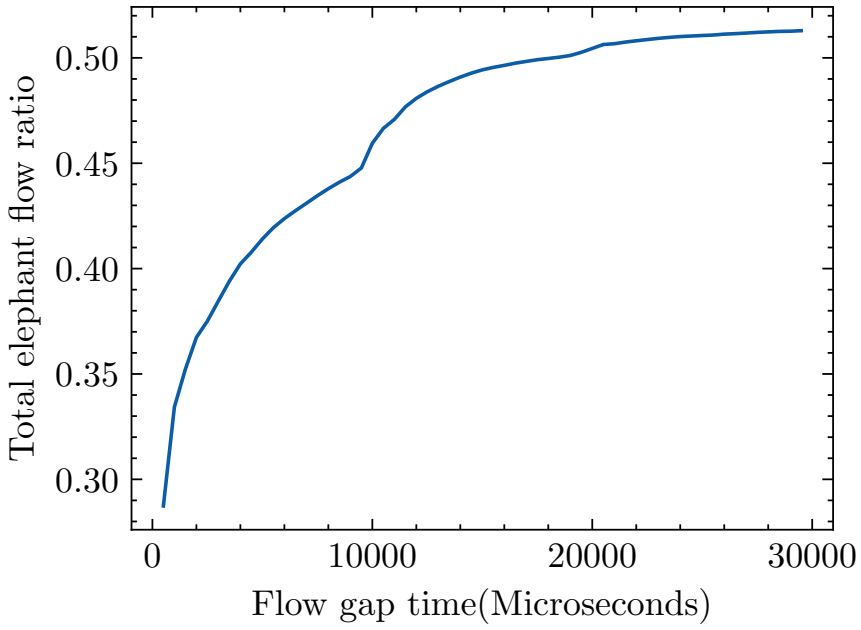


Figure 4.5: SVM Model training on spark.

view of the next flow size prediction problem.

On the other hand, our second approach involves treating the problem as a binary classification task. Here, we categorize flows into two classes based on the flow size. If the flow uses more than 10% of the total bandwidth [12], it is called an elephant flow, and in the other case it is called a mice flow. In this setup, we assign the label 1 to elephant flows and 0 to mice flows. Our model is then trained to predict these binary labels. It's important to see that in different flow time gaps the propotion of elephant flows and mice flows might not be equal, which is called the dataset class imbalance. We name this approach as the *Classification-based* view of the next flow size prediction problem.

The Figure 4.5 illustrates the proportion of elephant flows; for instance, at 5000 microseconds, 40% of the total flows are elephant flows.

After the labeling stage, our dataset is ready and we can use it as an input to our model.

Table 4.1: List of features used for next flow prediction. The features with (*) are only included in the *extended feature set*.

Feature Name	**Description**	**Type**
start_time	Start time of the flow	Network related
end_time	End time of the flow	Network related
gap	Time gap between the end of the previous flow and start of the current flow	Network related
neworkin	Incoming network traffic (bytes) from start of the workload until the start of the flow	Network Related
neworkout	Outgoing network traffic (bytes) from start of the workload until the start of the flow	Network Related
sendmsg	Trace of `sendmsg` function calls, consisting of time of function call and the total size of the sent buffer	Event Traces
src_ip	IP Address of the source of the flow	Network Related
dest_ip	IP Address of the destination of the flow	Network Related
disk_read	Data read from disk (bytes) from start of the workload until the start of the flow	Polled Trace
disk_write	Data written to disk (bytes) from start of the workload until the start of the flow	Polled Trace

Continued on next page

Table 4.1 – continued from previous page

Feature Name	Description	Type
memory	Virtual memory used (bytes) by the workload process	Polled Traces
memavailable	Total available host memory (bytes)	Polled Traces
s_time	The duration of this process scheduled in kernel mode, measured in clock ticks.	Polled Traces
data_memory*	Data memory used (bytes) by the workload process	Polled Traces
rss_memory*	Physical memory used (bytes) by the workload process	Polled Traces
u_time*	The duration of this process scheduled in user mode, measured in clock ticks.	Polled Traces
cpu_allocations*	Trace of `malloc` function calls, consisting of time of function call and the total allocated size	Event Traces
src_port*	Port of the source of the flow	Network Related
dest_port*	Port of the destination of the flow	Network Related
tcp_sendmsg*	Trace of `tcp_sendmsg` function calls, consisting of time of function call and the total size of the sent buffer	Event Traces
write*	Trace of `write` function calls, consisting of time of function call and the total size of the written buffer	Event Traces

Continued on next page

Table 4.1 – continued from previous page

Feature Name	Description	Type
sendto*	Trace of `sendto` function calls, consisting of time of function call and the total size of the sent buffer	Event Traces
kcache*	Trace of `kmem_cache_alloc` function calls, consisting of time of function call and the total size of the allocated buffer	Event Traces
cuda_allocations*	Trace of GPU coppied or allocated memory using CUDA library using `cudaMalloc`, `cudaMemcpy`, `cudaMallocAsync`	Event Traces
cuda_collective*	Trace of `ncclAllReduce` function calls, consisting of time of function call and communicated buffer size	Event Traces

4.3.2 Overhead of Trace Collection

One significant challenge encountered in our study is the massive overhead associated with collecting all the traces necessary for flow prediction. The process of gathering packet traces, polled system traces, and event-based traces from the host can introduce substantial computational and storage burdens. This overhead is impractical for a real-world implementation, where efficiency and minimal impact on system performance are critical.

To mitigate this issue, a robust feature engineering process is essential. By identifying and selecting the most impactful features for flow prediction, we can significantly reduce the

volume of data that needs to be collected. This process not only enhances the model's performance but also makes the trace collection process more manageable. Detailed discussions on how feature engineering helps in identifying the best traces to collect are provided in the section 5.5.

Furthermore, employing more advanced monitoring techniques can alleviate the overhead problem. Integrating trace collection processes within the hypervisor or utilizing specialized hardware for monitoring purposes can reduce the strain on the host system. These methods offer a more efficient and scalable solution for trace collection, paving the way for practical implementations in larger and more complex network environments.

4.4 Flow Size Prediction Machine Learning Model

For our training dataset, outlined in section 4.3, we structured the data in a table format where each row represents a flow, including specific features of that flow and the features of the previous five flows as historical context. The goal is to predict the size of the next flow, either as an exact value in a regression task or as a category (elephant or mice) in a classification task, based on the given features. The process of training our model to make these predictions is detailed in Figure 4.6.

We employed a Gradient Boosting Decision Trees (GBDT) model [17] for our predictive modeling task, using the CatBoost [31] library as an open-source implementation. GBDT models are built through iterative refinement in multiple boosting rounds. In each boosting round t, a new predictor (decision tree) is incorporated into the model to minimize the prediction error from the previous round $(t-1)$. The extent of prediction error is quantified

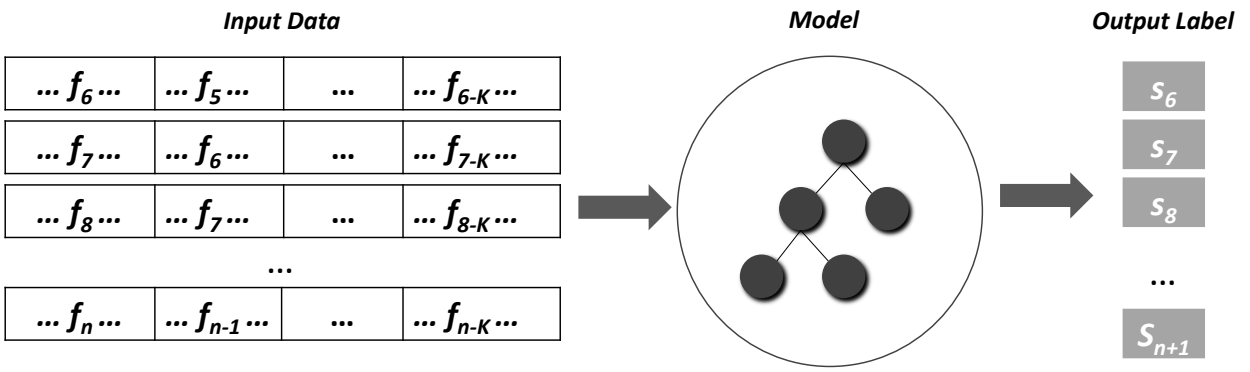


Figure 4.6: Model training procedure.

by a loss function tailored to the specific problem addressed by the model. The output of the model is calculated by aggregating the predictions of all of the predictors in the model.

For our use case CatBoost models are preferred over deep neural network (DNN) models, which often have longer inference time and require larger datasets, extensive optimization, and hyperparameter tuning. Additionally, the explainability of CatBoost models provides valuable insights that are not as easily obtainable with DNN models, making them more suitable for our specific use case in flow size prediction.

In the *regression-based* approach of the next flow size prediction, we used Mean Squared Error (MSE) as the objective function which is defined in the section 2.3.5. In the *classification-based* approach of the next flow size prediction we use the Log loss function:

$$-\sum_{i=1}^{N}(y_i \log(\hat{y_i}) + (1 - y_i) \log(1 - \hat{y_i})) \tag{4.1}$$

where N is the number of the instances in the dataset, y_i is the actual flow class and $\hat{y_i}$ is the predicted flow class by the model.

We used the maximum number of 1000 boosting iterations, and maximum depth of 6 for

each decision tree.

Chapter 5

Evaluations

In this section, we outline our evaluations for flow size prediction. We begin by detailing the setup of our evaluations. Following this, we present our results in two distinct settings: regression-based and classification-based predictions. We proceed to compare these models to discern their respective efficacies. Additionally, we conduct a feature importance analysis to identify the most critical features in each workload.

5.1 Evironment Setup

In order to have a comprehensive understanding of our results, it is necessary to analyze various metrics. We use R^2, MAPE, MSE, and F1 score to evaluate the performance of flow size prediction.

For all experiments, we allocated 70% of our collected dataset for training and the remaining 30% for testing. All experiments are done for different workloads and a range of flow time gaps, spanning from 500 microseconds to 30000 microseconds.

Experiments consider the extended feature set and the limited feature set. The limited feature set is derived from the FLUX[10], which includes various statistics extracted about host and network activities of a workload. For the extended feature set, we added event-based features and additional features on top, aiming to capture a more comprehensive range of system activities and interactions.

The Spark applications, *i.e.*, Page Rank algorithm, SVM and K-Means models, were run on Amazon EMR [50] comprising three machines, each equipped with 8 CPU cores and 16 GB of memory. We run PyTorch GPT Model on an AWS cluster consisting of three AWS EC2 nodes [63]. Each EC2 node is equipped with one NVIDIA T4 GPU. Finally, Tensorflow MNIST Model was run on a local cluster consisting of four machines, with each node having 8 CPU cores. A detailed description of workloads can be found in Section 3.

We evaluate the model in both regression-based and classification-based settings. In both of these settings, the model is trained to predict the exact size or the label of the next flow in our workloads across 500 microseconds to 30000 microseconds flow time gaps. We use MSE, R^2 and MAPE for the model evaluation in regression-based setting and F1*score* for the assessment of the model in classification-based setting. We compare the accuracy of the model trained with the limited feature set with the model trained with the extended feature set.

5.2 Flow Size Prediction

In this section, we present the results of our next flow size prediction using regression-based models. The regression-based model performance on PyTorch GPT Model workload is shown

in Figure 5.1. The following observations are in order. First, we can observe that the R^2 score for the extended feature set is 0.6 higher than the score of the limited feature set. Also, the MSE score of the model trained with extended feature set is improved by at least 0.2 in most of the flow time gaps. Improved MSE and R^2 suggests that the extended feature set can improve the model's ability to predict the size of the next flow. Second, the MAPE score for the flow size prediction with a extended feature set has an error ranging from 2% in 10000 microseconds to 3% in 30000 microseconds. The low percentage error in this setup shows that the next flow size prediction is feasible with high accuracy.

In Figure 5.1, an increasing R^2 value with larger flow time gaps suggests that the variability in flow size is being captured more significantly by the model with extended feature set. This means that the model's predictions are closely following the variations in the actual flow sizes but it doesn't necessarily mean that the predictions are accurate, only that they are varying in a way that's proportional to the actual flow size's variance. The increase in MAPE within the 500 and 5000 microseconds flow time gap indicates that as the flow time gap increases, the model finds it more difficult to accurately predict the next flow size. In this range, there's more variability in the flows that the model hasn't learned to capture. The decrease in MAPE in 5000 to near 14000 micorseconds flow time gap, suggests that the model is better at capturing the pattern of flows within this time gap range. It shows a certain regularity in the flow sizes that the model can learn. The subsequent increase in MAPE in flow gap times larger than 14000 microseconds indicate that the model has difficulty due to increased variability or noise in the flow sizes.

The regression-based performance on Tensorflow MNIST Model workload is shown in Figure 5.2. Although the achieved MAPE (12% in worst case) shows that the next flow

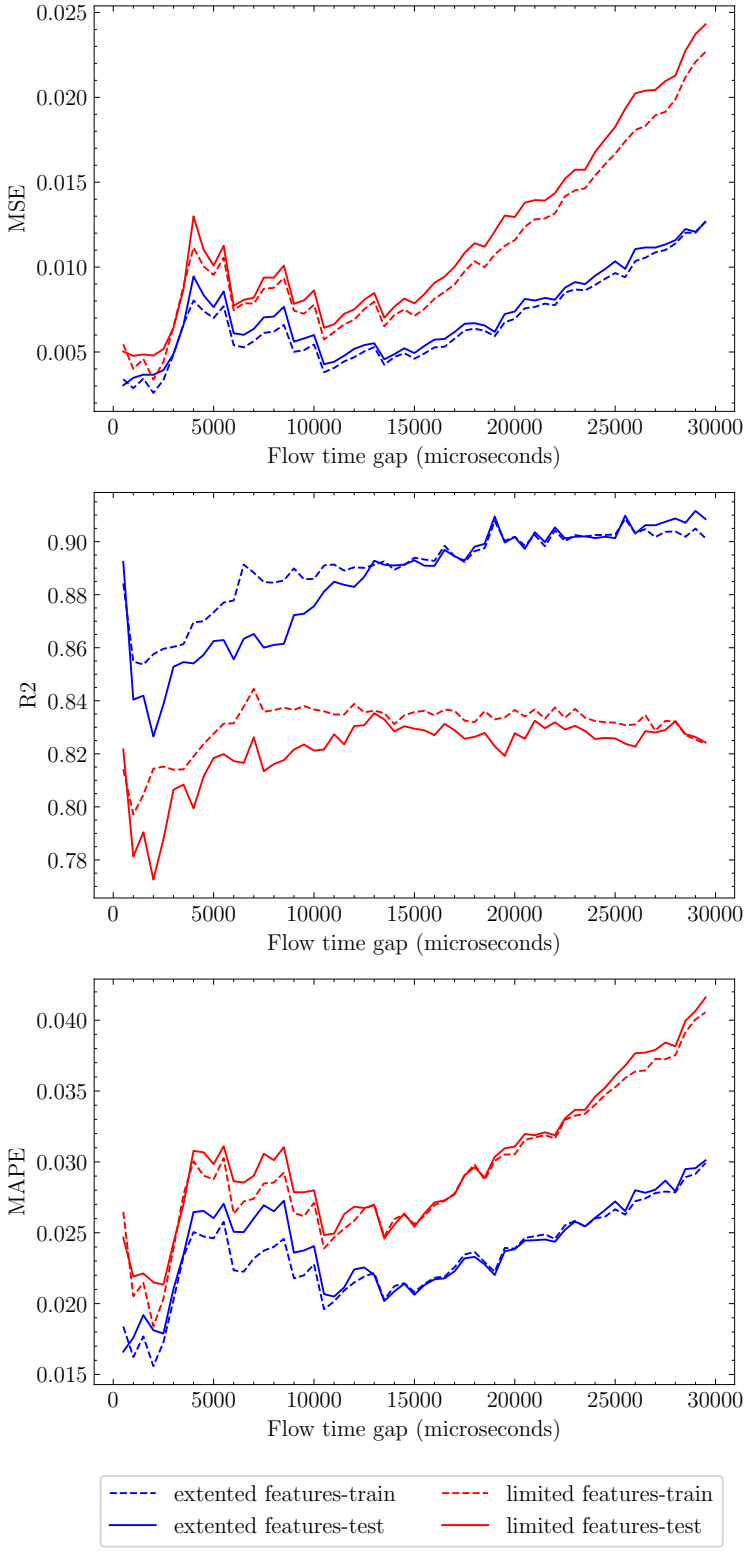


Figure 5.1: Regression-based model results on PyTorch GPT Model workload.

prediction is feasible with good accuracy, we can see no improvements in MSE and minor improvements in R^2 (in flow time gaps less than 5000 microseconds) with the extended feature set compared to the limited feature set. This show the features added in the extended feature set does not help the model to increase the prediction accuracy in this workload.

We can see the R^2 score of the extended feature set is 0.2 higher than the R^2 score of the limited feature set in SVM and KMeans models on spark workload as illustrated in Figure 5.3 and Figure 5.4. Thus, using more features in training workloads greatly benefits the model, resulting in significant score improvements. Conversely, the extended feature set did not yield better results in the PageRank workload. Despite this, the low MAPE (near 0% in all flow time gaps) score in the PageRank workload suggests that the next flow size prediction in this context is highly accurate and exhibits minimal error, indicating a high level of predictability and stability in flow size generation for the PageRank workload.

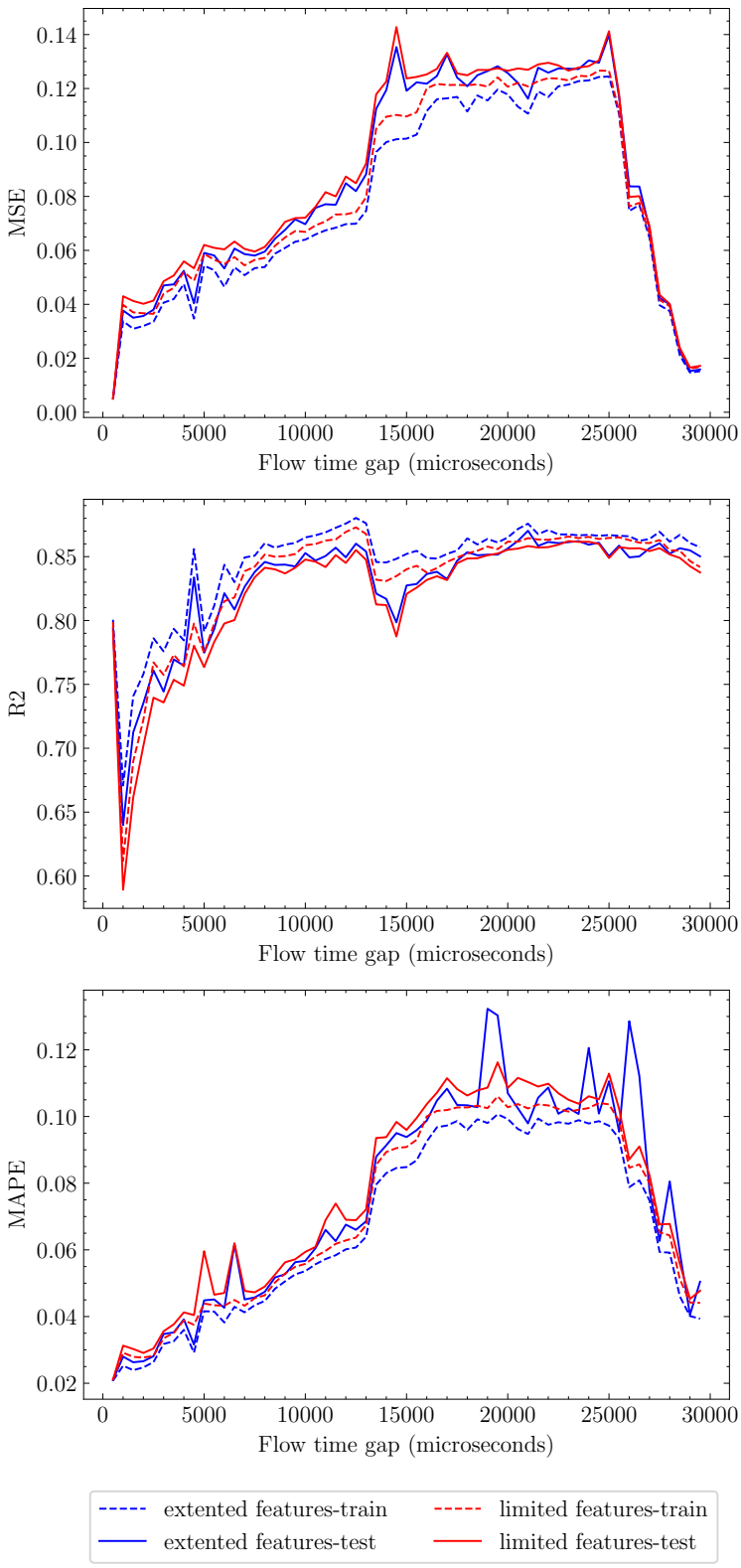


Figure 5.2: Regression-based model results for the Tensorflow MNIST Model workload.

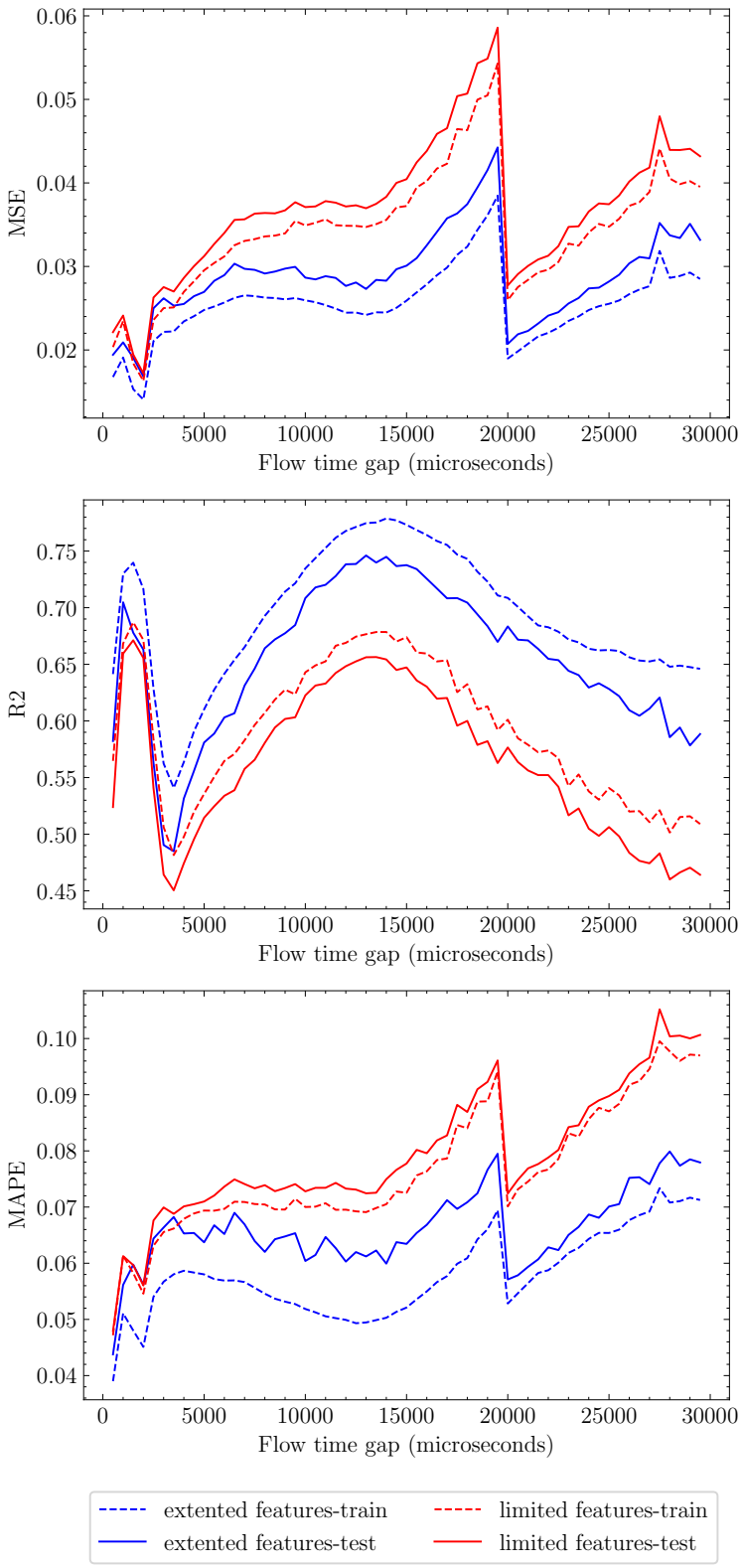


Figure 5.3: Kmeans Model training on spark.

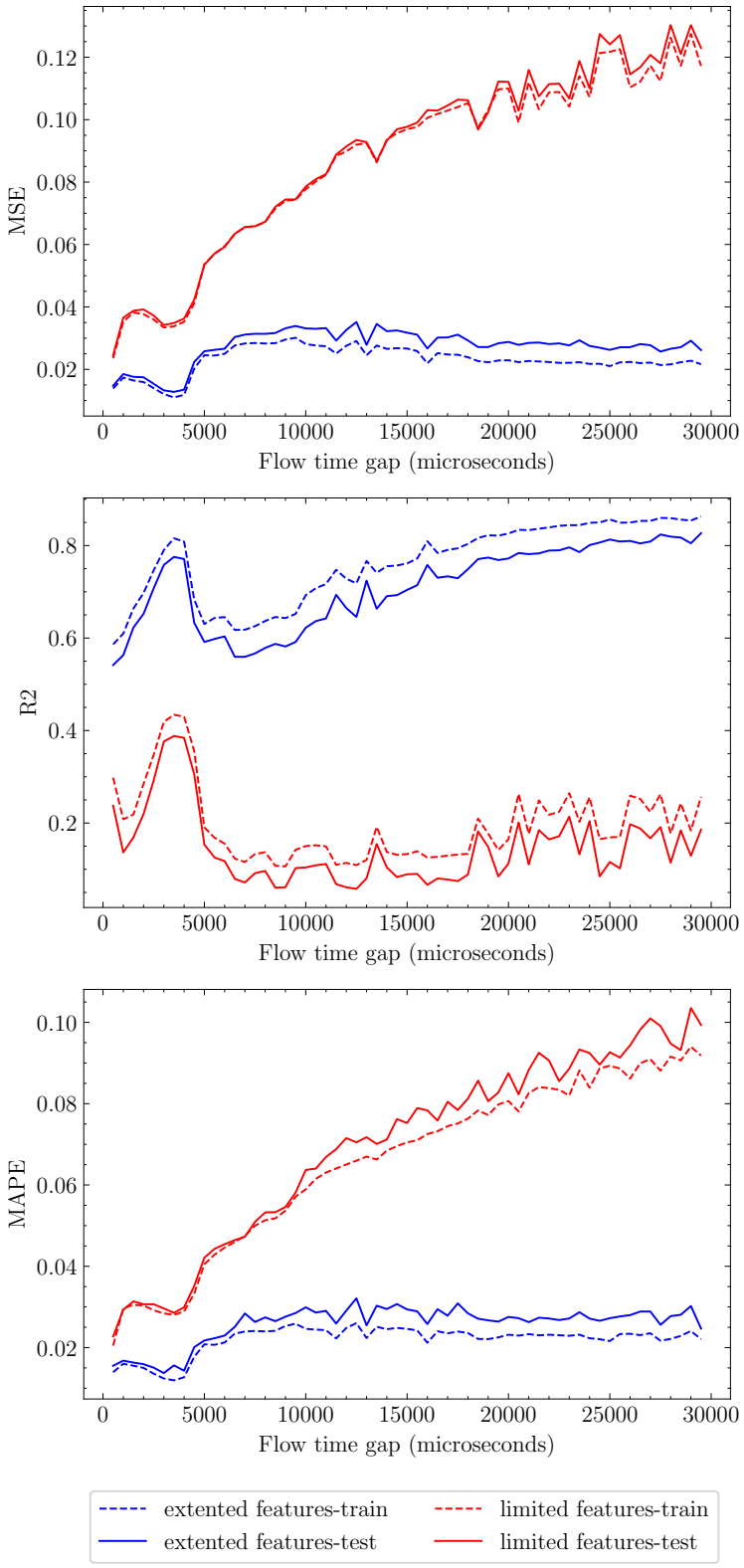


Figure 5.4: SVM Model training on spark.

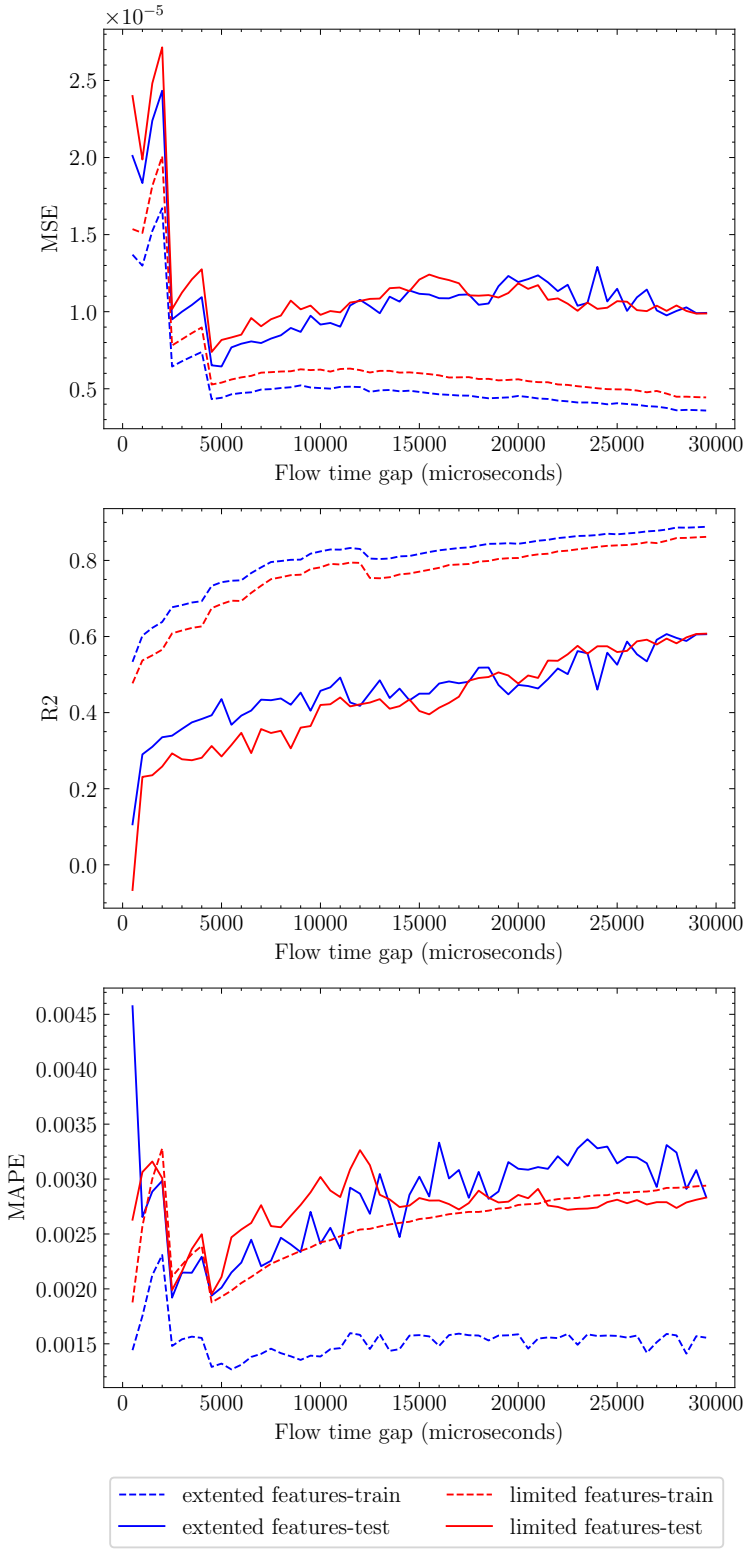


Figure 5.5: PageRank algorithm on spark.

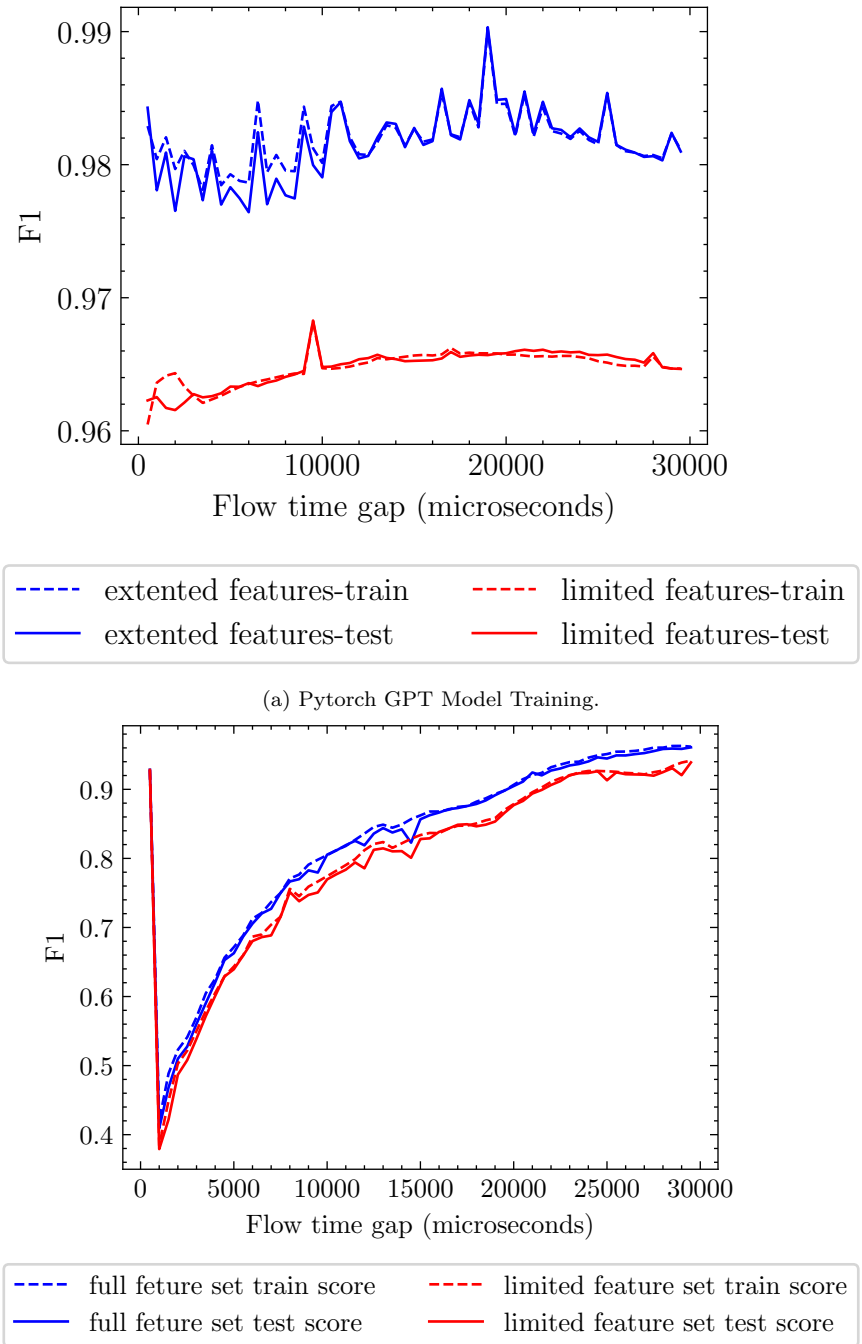
5.3 Flow Size Classification

In this section, we present the results of our next flow size classification using classification-based models. The classification-based model performance on PyTorch GPT Model workload is depicted in Figure 5.6(a). From the figure, we can observe that the extended feature set has a score of 0.98 for most flow time gaps, which has 0.02 improvement over the limited feature set. The results for the Tensorflow MNIST Model workload for classification-based model is shown in Figure 5.6(b). We can see that, in both classification-based models, the extended feature set does not significantly improve the model's F1 score.

In Figure 5.7 we can observe that the next flow classification for spark workloads, is improved by the extended feature set. Specifically, 0.15 point improvement for Kmeans model workload in all flow time gaps, 0.15 point improvement for SVM model workload for 10000 microseconds and above flow time gaps, and 0.1 point improvement for Pagerank workload in all flow time gaps shows the extended feature set improvement over the limited feature set.

5.4 Classification with Regression-based Models

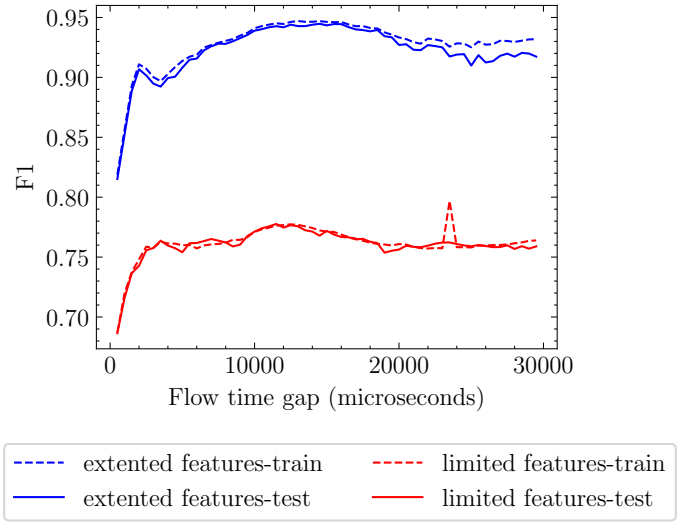
An alternative strategy for the classification-based setting involves training the model in a regression-based setting. To assess the regression-based trained model in the classification-based setting, the elephant/mice size threshold is applied to identify the next flow label. If the predicted size of the next flow is greater than the threshold, it is labeled as an elephant flow; otherwise, it is considered a mice flow. In Figure 5.8, the F1 score for the classification-based setting evaluation of the two models is shown. Notably, the model trained in the



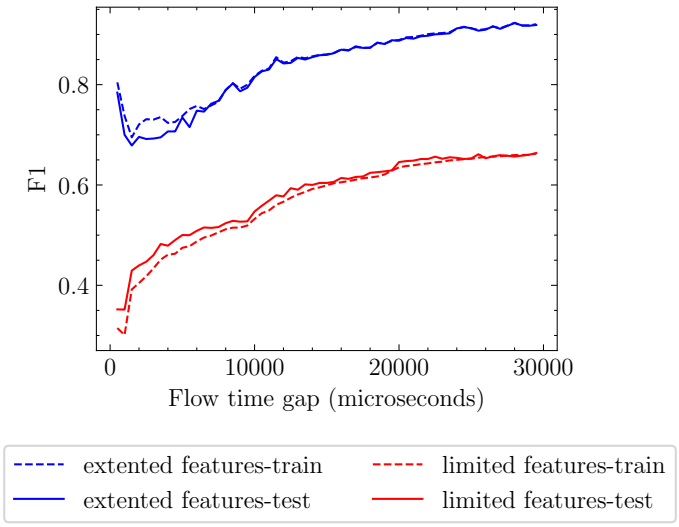
(a) Pytorch GPT Model Training.

(b) Tensorflow Model Training.

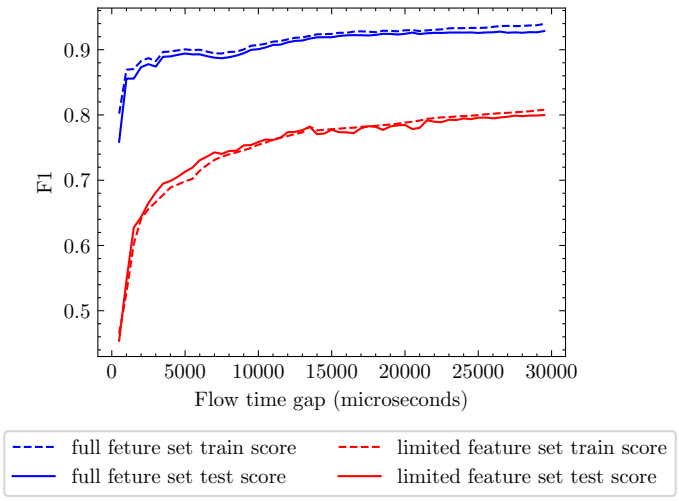
Figure 5.6: Deep Learning Workloads Flow Classification F1 score.



(a) Kmeans Model training on spark.



(b) SVM Model training on spark.



(c) PageRank on spark.

Figure 5.7: Spark workloads Flow Classification F1 score.

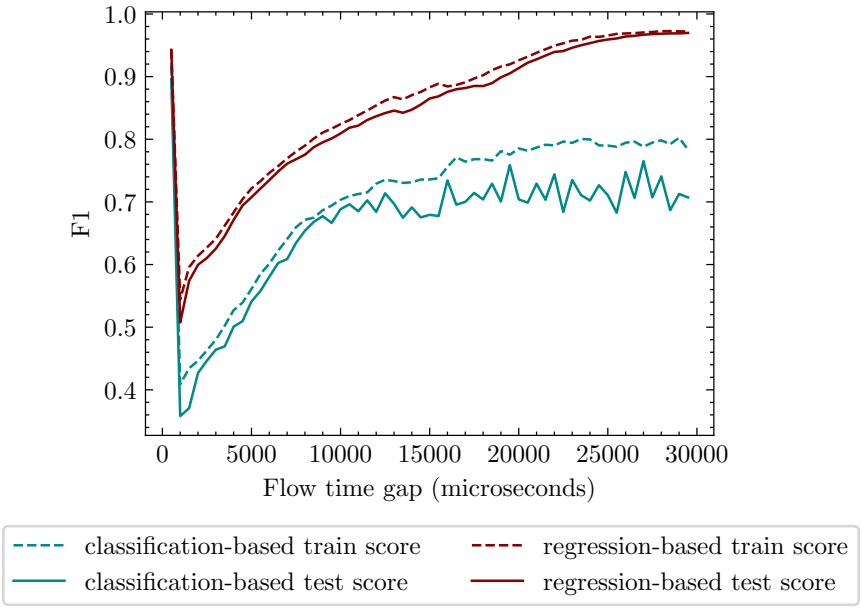


Figure 5.8: Pytorch GPT Model Training flow classification results.

classification-based setting consistently outperforms the model trained in the regression-based setting by a margin of at least 0.1, and this difference increases to 0.25 in larger flow time gaps. The improved performance of the classification-based setting trained model can be attributed to its alignment with the evaluation goal. In contrast, the regression-based trained model is optimized with a different objective, specifically, minimizing the Mean Squared Error (MSE). While the F1 score of the regression-based model is comparatively lower, a notable advantage of training the model in a regression-based setting is its versatility—it can be utilized for both regression and classification tasks. On the other hand, the classification-based setting trained model can only be used for the classification task.

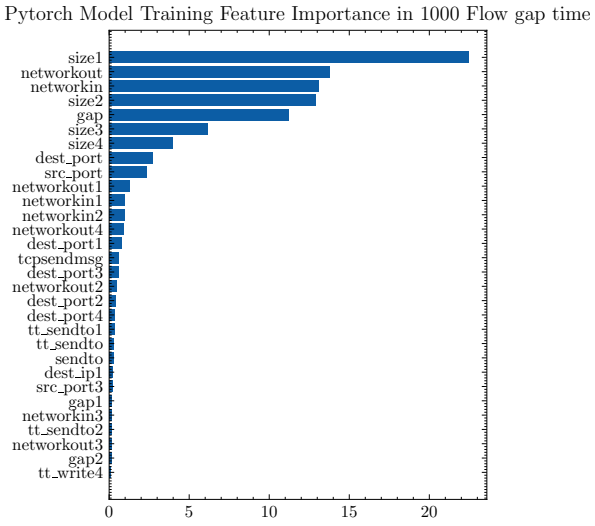
5.5 Flow Size Prediction Feature Importance Analysis

Figure 5.9 presents the feature importances of the model trained for the Pytorch GPT Model workload for different time gaps. From the figure, we observe that the network-related fea-

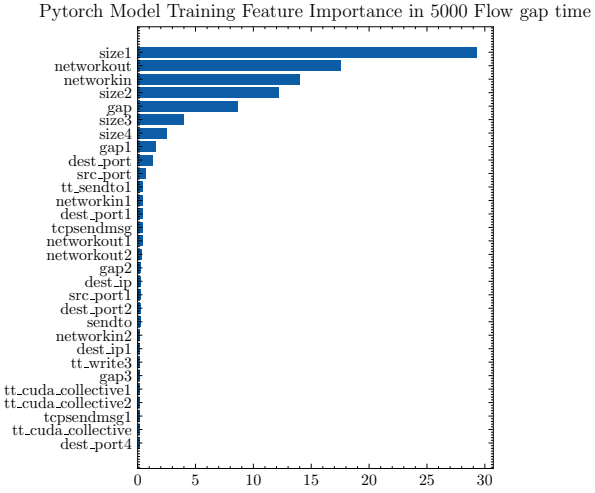
tures (*i.e.*, networkin, networkout) are the most important features for this workload. Additionally, we can see that as the flow gap time increases, the GPU-related event-based features become more important. For instance, the importance of *cuda_collective* feature, corresponding to the *ncclAllReduce* function call event, increases as the flow time gap increases from 500 microseconds to 30000 microseconds. Also in Figure 5.10 the SHAP values of the most impactful features is shown.

In distributed deep learning and similar computational workloads, different features can carry varying degrees of importance depending on the specific characteristics and demands of the workload. When considering the feature importances for the Pytorch GPT Model workload across different flow gap times, we notice that network-related features are consistently crucial. This indicates that the transmission of data across the network is a significant factor in predicting flow sizes in such a setup, likely due to the need for data synchronization between different nodes during model training.

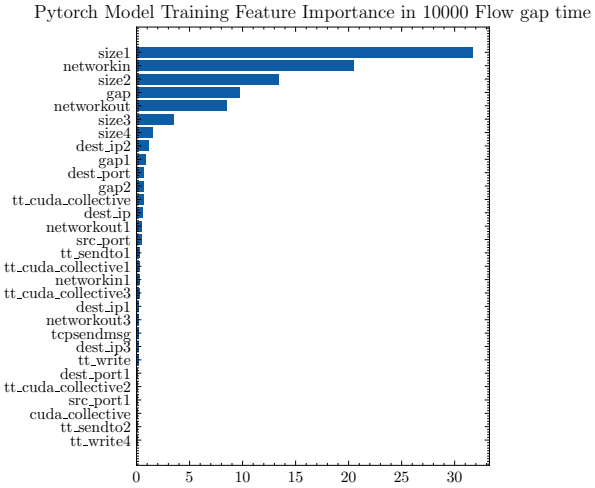
As the flow gap time increases, the model begins to weigh GPU-related event-based features more heavily, particularly the *cuda_collective* feature associated with the ncclAllReduce function call event. This shift suggests that as the time gap extends, the aggregation of data across GPUs, which is essential for synchronizing the model's state across the distributed system, becomes a more influential factor. The increase in the flow gap time could represent a scaling in the model's complexity or the volume of data being processed, thereby making GPU synchronization a critical component of accurate flow size prediction.



(a) Feature importance for 1000 microseconds.

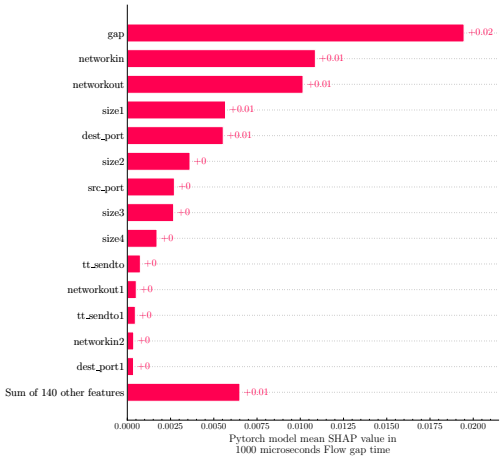


(b) Feature importance for 5000 microseconds.

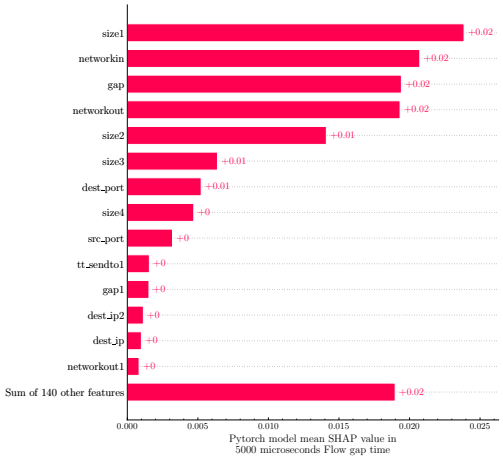


(c) Feature importance for 10000 microseconds.

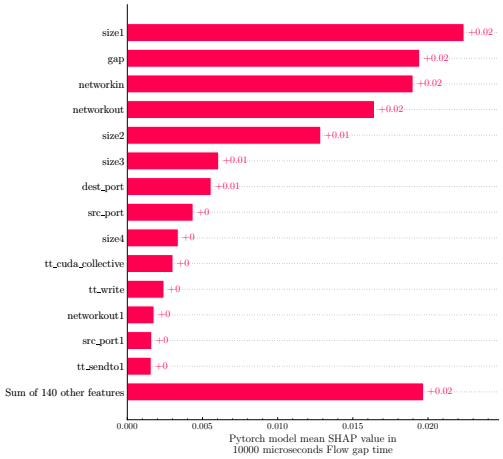
Figure 5.9: Feature Importances of features in PyTorch Model multi GPU training. In PyTorch workload, the network related features are most important for the model in determining the future flow size.



(a) Mean SHAP value for 1000 microseconds.



(b) Mean SHAP value for 5000 microseconds.



(c) Mean SHAP value for 10000 microseconds.

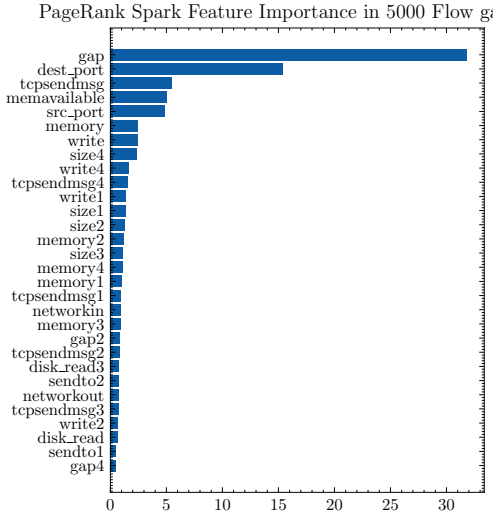
Figure 5.10: Mean SHAP Values of features in Pytorch Model multi GPU training. In PyTorch workload, the network related features are most important for the model in determining the future flow size.

In Figure 5.11, the feature importances for flow prediction on Spark workloads with a 5000 microseconds flow gap time are depicted. Notably, event-based features such as *tcpsendmsg* (*i.e.*,, the *tcp_sendmsg* system call event), *cpu_allocation* (*i.e.*,, the *malloc* call event), and others prove to be highly valuable for the model. Also in Figure 5.12 the SHAP values of the most impactful features is shown.

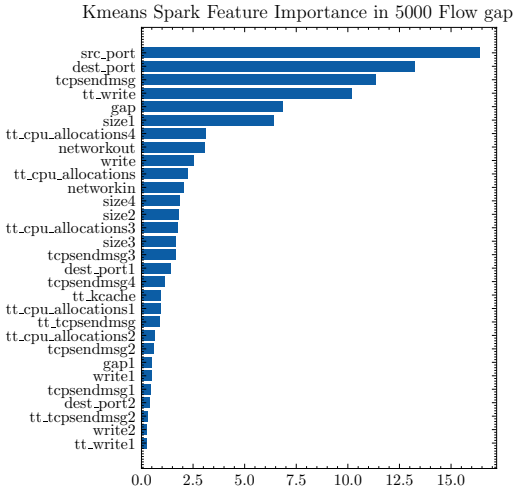
For the Spark workloads, event-based features like *tcpsendmsg* and *cpu_allocation* stand out as particularly important. These features are related to system calls for sending messages over TCP and memory allocation, respectively. Their significance could imply that network communication and memory management are critical for the performance of the Spark tasks, which might involve significant shuffling of data over the network and memory operations due to the nature of the MapReduce paradigm Spark builds upon.

Overall, the most important features highlight the operational bottlenecks or key activities that dominate the workload's behavior. Network features being pivotal for a Pytorch model emphasizes communication overhead in model synchronization, whereas system call-related features for Spark point towards network IO and memory allocation as dominant factors in the distributed data processing tasks. Understanding these nuances is essential for optimizing system performance and tuning the models for precise flow prediction.

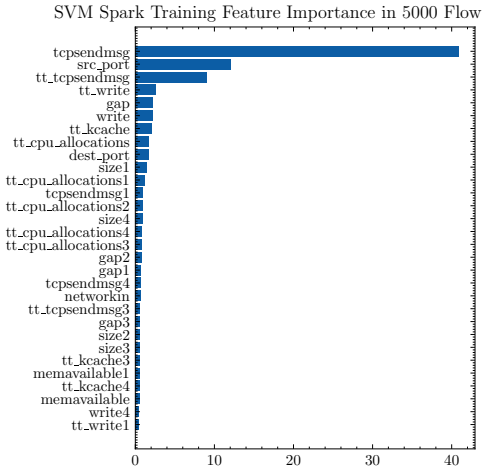
Recording traces concurrently with the running workload incurs host overhead; thus, it is advisable to minimize the number of recorded traces. The optimal traces for recording can be identified based on the feature importances in each workload. We utilized the top 5 features for flow size prediction in regression settings for each workload. Table 5.1 displays the R^2 scores for flow prediction using the selected features. Notably, the maximum R^2 drop across all workloads is 2%, underscoring the effectiveness of feature selection.



(a) Feature importance for PageRank.

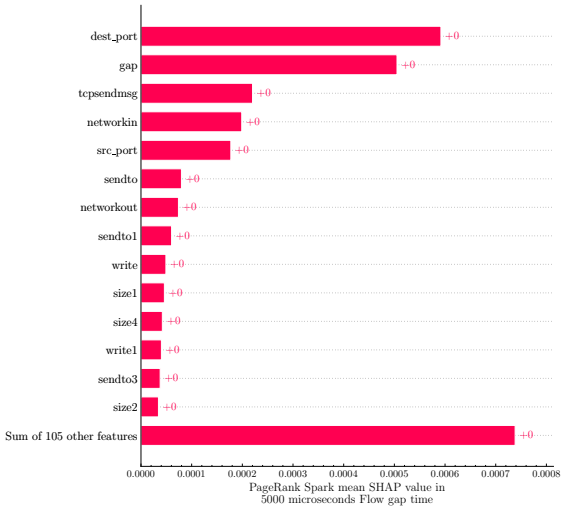


(b) Feature importance for KMeans.

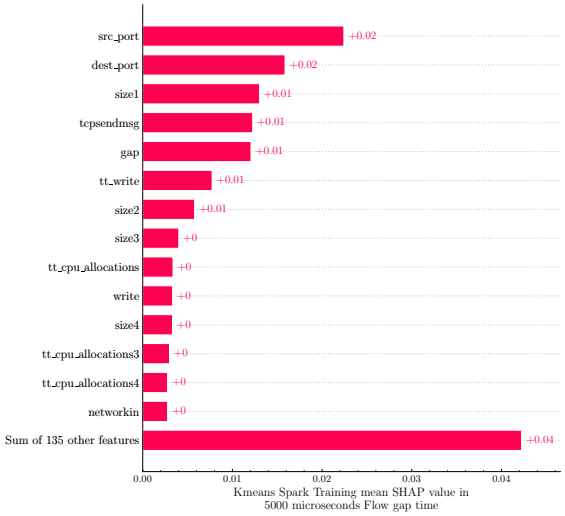


(c) Feature importance for SVM.

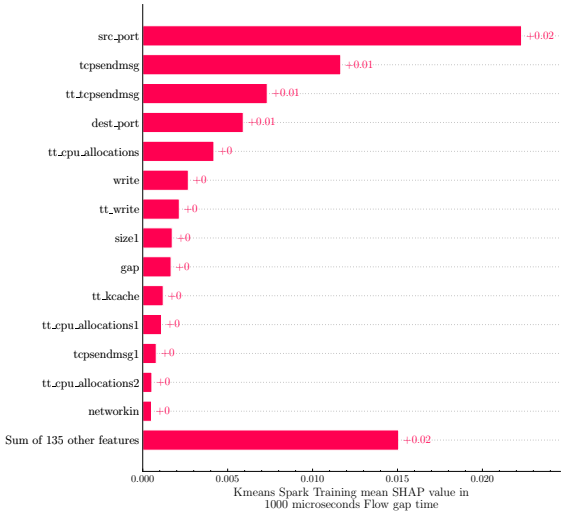
Figure 5.11: Feature Importances of features in Spark Workloads for 5000 microseconds flow gap time. In Spark workloads, the event-based features are most important for the model in determining the future flow size.



(a) Mean SHAP value for PageRank.



(b) Mean SHAP value for KMeans.



(c) Mean SHAP value for SVM.

Figure 5.12: Mean SHAP Values of features in Spark Workloads for 5000 microseconds flow gap time. In Spark workloads, the event-based features are most important for the model in determining the future flow size.

Flow Gap Times (microseconds)		1000	5000	10000	15000
Pytorch Model	All Features	0.840	0.862	0.875	0.892
	Top-5 Features	0.826	0.839	0.848	0.855
Tensorflow Model	All Features	0.639	0.774	0.852	0.827
	Top-5 Features	0.603	0.774	0.853	0.829
Spark Kmeans	All Features	0.739	0.590	0.723	0.756
	Top-5 Features	0.745	0.594	0.725	0.761
Spark SVM	All Features	0.563	0.591	0.623	0.704
	Top-5 Features	0.572	0.60	0.634	0.70
Spark PageRank	All Features	0.290	0.435	0.467	0.449
	Top-5 Features	0.281	0.435	0.463	0.449

Table 5.1: R^2 scores of flow prediction of the flows generated by different workloads by using all features, or the best features of each workload.

5.6 Flow Size Prediction Result Discussion

In this section, we aim to understand why the trend in our results changes across different flow time gaps. To do this, we examine the behavior of the PyTorch GPT model as an example, which helps us identify the underlying factors influencing predictive performance. Other workloads should follow the same way of thinking, enabling a comprehensive understanding of the impact of flow time gaps on prediction accuracy.

We conduct an analysis of the observed trends in Mean Squared Error (MSE) in the results of Pytorch Model multi-GPU training, as illustrated in Figure 5.1. To delve into this trend, we investigate the relationship between the occurrences of `ncclAllReduce` calls and the variability observed in the generated flows across different flow time gaps. The `ncclAllReduce` function, a CUDA library function employed by GPU workloads, is invoked when hosts engaged in deep learning training seek to exchange information regarding the subsequent update to the deep learning model. Notably, `ncclAllReduce` accounts for a

significant portion of network activities among the involved hosts. Figure 5.13 presents the histogram depicting the time intervals between consecutive `ncclAllReduce` calls.

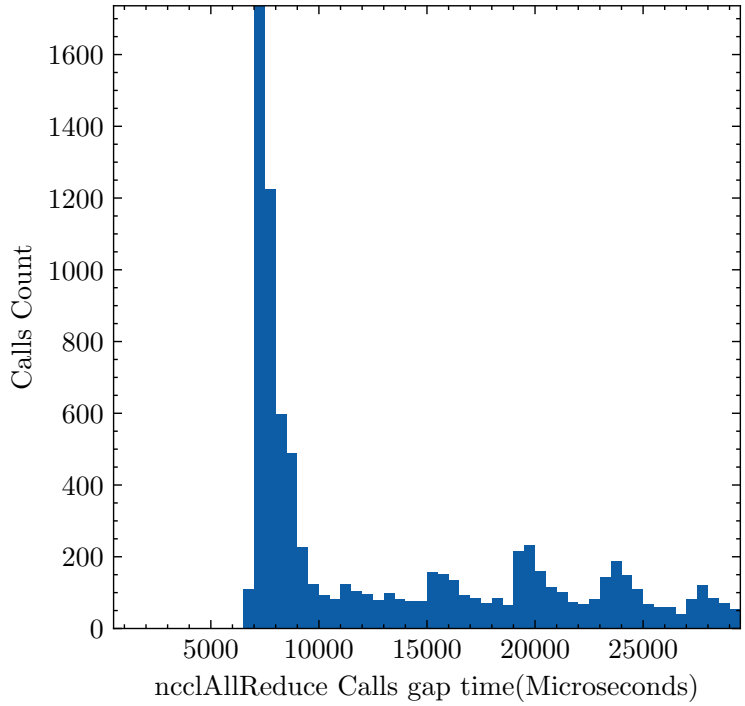


Figure 5.13: Histogram of time gaps between *ncclAllReduce* calls.

In order to compare the variability of genereted flows in different flow time gaps, we examine the coefficient of variation of flow sizes in each flow time gap. The coefficient of variation is calculated as the ratio of the standard deviation of flow sizes to the mean of flow sizes corresponding to a specific flow time gap. The coefficient of variation for flow sizes is depicted in Figure 5.14. It illustrates a reduction in variability at the 10000 microseconds flow time gap, indicating a more regular workload behavior at this interval, therefore the loss of the trained model (*i.e.*, MSE) decreases compared to other flow time gaps. As the flow gap times increase, the variability tends to rise, suggesting less predictability in the workload pattern. As the flow time gap increases beyond 10000 microseconds, the flows generated by individual `ncclAllReduce` calls become more intertwined and aggregated, introducing

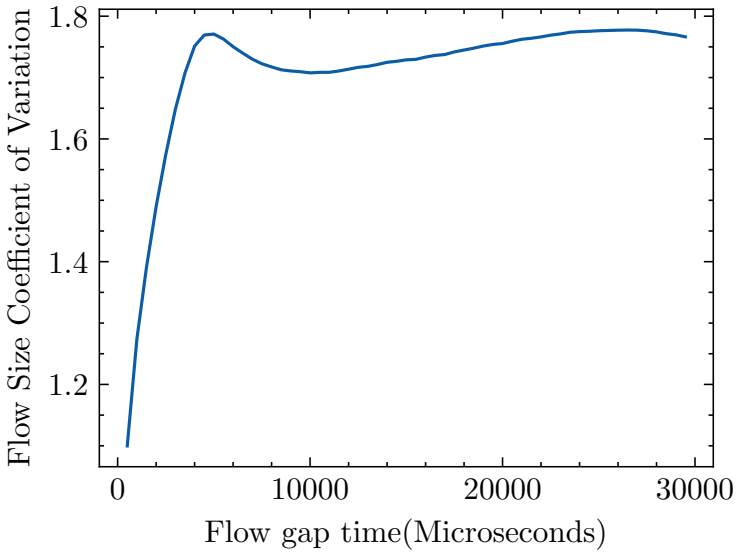


Figure 5.14: Pytorch Model multi GPU training flow size Coefficient of Variation.

greater variability in the generated flow sizes. It's noteworthy that at flow gap times shorter than 10000 microseconds, there is a sudden increase in the coefficient of variation and higher MSE. This suggests more rapid and unpredictable flow generation in the workload. Since there are no two `ncclAllReduce` calls that have a gap less than 6000 microseconds (As shown in Figure 5.13, the impact of the features related to this function is near zero before the 6000 flow time gap. The jump in coefficient of variation in around 6000 microseconds suggest that the generated flows in that flow time gap, does not follow a consistant pattern. This trend analysis can be extended to other workloads by a simillar procedure.

Chapter 6

Conclusion and Future Works

6.1 Conclusion

In our research, we presented a machine learning approach to accurately forecast the size and categorize the next flow (i.e., Elephant/Mice). This prediction relies on features extracted from the hosts engaged in the workload, achieving a high level of accuracy.

We incorporated a novel set of features known as event-based features to enhance the accuracy of our model predictions. Through our feature evaluation, we found out that the event-based features have higher feature importance than the polled features but the network related features proved to be the most important features.

We designed a data collector module designed to gather necessary data for the model using various tools (*e.g.*,*eBPF*, */proc*, *etc.*). Importantly, this data collection process doesn't require any modifications to the workload application code, making it a convenient choice for network operators.

We conducted an assessment of our model using five distinct target workloads on public

and local testbeds. We investigated the influence of various flow gap times (ranging from 500 microseconds to 30000 microseconds) on the accuracy of the model predictions. The effect of flow time gap on the accuracy of the model is determined by the workload behaviour, to show this we analysed the Pytorch GPT Model workload's behaviour accross the flow time gaps.

6.2 Thesis Summary

This thesis can be summerized as follows:

- Chapter 1 introduces the research motivation, emphasizing the significance of predicting flow sizes in network systems. It also reviews current flow prediction techniques and outlines the thesis objectives.

- Chapter 2 offers a detailed background on key subjects relevant to this study. It covers the technical details of decision trees, ensemble learning, and XGBoost, setting the foundation for the methodologies applied in this research. The chapter further explores system monitoring tools such as eBPF, PF_RING, and the Linux proc directory, along with an in-depth discussion on the evaluation metrics used. Also the section 2.2 includes a literature review that examines existing flow prediction systems, identifies their shortcomings, and outlines the research gaps this thesis seeks to fill.

- Chapter 3 outlines the various workloads utilized in this research, starting with an introduction and then detailing the use of AWS for the infrastructure. It elaborates on deep learning workloads, including distributed deep learning training with discussions

on both PyTorch and TensorFlow frameworks. Furthermore, it explores the integration of Spark and Hadoop in the research, specifically examining Spark KMeans, Spark PageRank, and Spark SVM workloads.

- Chapter 4 outlines the methodologies and tools utilized for gathering data and constructing the datasets crucial for this study. It begins with a general overview, then delves into the trace collection process, categorizing the traces into packet, polling, and event traces. The chapter also explains the construction of the datasets. Finally, it presents the model used in this research.

- Chapter 5 delves into the evaluation of the research, beginning with the setup of the workloads used for testing. The chapter further segments into the core focus on flow size prediction, examining both regression-based approaches and classification methodologies. Additionally, it evaluates the importance of various features in the prediction process and concludes with an analysis of the results obtained from these methodologies.

6.3 Future Works

There are several areas in which this research work can be extended. The following can be considered for the future:

- **Enhancing trace collection efficiency.** Current methods, which operate on the host, may introduce overhead to running applications. Future iterations could explore more efficient monitoring software, integration with datacenter hypervisors, or the use

of dedicated hardware specifically for monitoring purposes.

- **Using different features and trace types.** Future research could significantly benefit from exploring a variety of features and trace types. This work utilized event-based, network-related, and polling traces, each sourced differently. There's potential for applying diverse feature engineering methods to these traces and incorporating more functions for event-based traces. Moreover, developing an automated framework to identify the most effective functions to trace for varying workloads could enhance the precision and applicability of the findings.

- **Different Modeling Approach.** By leveraging the proven utility of low-level traces, future studies could enhance models by examining more deeply the intricate interactions among these traces. Furthermore, integrating spatio-temporal characteristics of network nodes into these models could provide deeper insights. It's crucial that any new models developed maintain high processing speeds to remain applicable in low-latency network environments, ensuring both efficiency and effectiveness.

- **Extending to Larger Clusters.** Another avenue for future work involves extending our methodology to larger clusters. This would involve scaling our predictive models to handle the increased complexity and data volume associated with larger distributed systems, ensuring that our approaches remain efficient and effective in more extensive, real-world scenarios.

Bibliography

[1] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: a centralized "zero-queue" datacenter network," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 307–318, Aug. 2014.

[2] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: a hybrid electrical/optical switch architecture for modular data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 339–350, Aug. 2010.

[3] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 407–420.

[4] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: minimal near-optimal datacenter transport," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 435–446, Aug. 2013.

[5] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: a receiver-driven low-latency transport protocol using network priorities," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 221–235.

[6] Y. Yu, J. Wang, M. Song, and J. Song, "Network Traffic Prediction and Result Analysis Based on Seasonal ARIMA and Correlation Coefficient," in *2010 International Conference on Intelligent System Design and Engineering Application*, vol. 1, Oct. 2010, pp. 980–983.

[7] A. Bayati, K.-K. Nguyen, and M. Cheriet, "Gaussian process regression ensemble model for network traffic prediction," *IEEE Access*, vol. 8, pp. 176 540–176 554, 2020.

[8] M. Li, Y. Wang, Z. Wang, and H. Zheng, "A deep learning method based on an attention mechanism for wireless network traffic prediction," *Ad Hoc Networks*, vol. 107, p. 102258, Oct. 2020.

[9] X. Ma, B. Zheng, G. Jiang, and L. Liu, "Cellular Network Traffic Prediction Based on Correlation ConvLSTM and Self-Attention Network," *IEEE Communications Letters*, vol. 27, no. 7, pp. 1909–1912, Jul. 2023.

[10] V. Dukic, S. A. Jyothi, B. Karlas, M. Owaida, C. Zhang, and A. Singla, "Is advance knowledge of flow sizes a plausible assumption?" in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 565–580.

[11] S. H. Mortazavi, A. Munir, M. M. Bahnasy, H. Dong, S. Wang, and Y. Ganjali, "Early-Bird: automating application signalling for network application integration in data-centers," in *Proceedings of the ACM SIGCOMM Workshop on Network-Application Integration*, ser. NAI '22. New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 40–45.

[12] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10. USA: USENIX Association, Apr. 2010, p. 19.

[13] A. Lazaris and V. K. Prasanna, "Deep learning models for aggregated network traffic prediction," in *2019 15th International Conference on Network and Service Management (CNSM)*, 2019, pp. 1–5.

[14] M. Balanici and S. Pachnicke, "Server Traffic Prediction Using Machine Learning for Optical Circuit Switching Scheduling," in *Photonic Networks; 20th ITG-Symposium*, May 2019, pp. 1–3.

[15] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *2011 Proceedings IEEE INFOCOM*, Apr. 2011, pp. 1629–1637.

[16] P. Poupart, Z. Chen, P. Jaini, F. Fung, H. Susanto, Y. Geng, L. Chen, K. Chen, and H. Jin, "Online flow size prediction for improved network routing," in *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, Nov. 2016, pp. 1–6.

[17] J. H. Friedman, "Greedy function approximation: A gradient boosting machine." *The Annals of Statistics*, vol. 29, no. 5, pp. 1189 – 1232, 2001.

[18] R. Xu, W. Li, K. Li, X. Zhou, and H. Qi, "DarkTE: Towards Dark Traffic Engineering in Data Center Networks with Ensemble Learning," in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, Jun. 2021, pp. 1–10, iSSN: 1548-615X.

[19] F. Han, Q. Li, K. Zhu, J. Zhou, Y. Jiang, Z. Qi, and F. Li, "LAFS: Learning-Based Application-Agnostic Flow Scheduling for Datacenters," in *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, Oct. 2021, pp. 1–8, iSSN: 2374-9628.

[20] "The llvm compiler infrastructure," accessed: 2024-02-29. [Online]. Available: https://llvm.org/

[21] "Bpf compiler collection (bcc)," accessed: 2024-03-23. [Online]. Available: https://github.com/iovisor/bcc

[22] "PF_ring," Aug. 2011, accessed: 2024-02-29. [Online]. Available: https://www.ntop.org/products/packet-capture/pf_ring/

[23] "Tcpdump & libpcap," accessed: 2024-02-29. [Online]. Available: https://www.tcpdump.org/

[24] C. J. S. R. O. Leo Breiman, Jerome Friedman, *Classification and Regression Trees*. Chapman and Hall/CRC, 1984.

[25] R. A. Fisher, "Iris," UCI Machine Learning Repository, 1988, DOI: https://doi.org/10.24432/C56C76.

[26] T. G. Dietterich, "Ensemble methods in machine learning," in *Proceedings of the First International Workshop on Multiple Classifier Systems*, ser. MCS '00. Berlin, Heidelberg: Springer-Verlag, 2000, p. 1–15.

[27] R. Polikar, "Ensemble Learning," in *Ensemble Machine Learning: Methods and Applications*, C. Zhang and Y. Ma, Eds. New York, NY: Springer, 2012, pp. 1–34.

[28] R. E. Schapire, "Explaining AdaBoost," in *Empirical Inference: Festschrift in Honor of Vladimir N. Vapnik*, B. Schölkopf, Z. Luo, and V. Vovk, Eds. Berlin, Heidelberg: Springer, 2013, pp. 37–52.

[29] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[30] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794.

[31] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "CatBoost: unbiased boosting with categorical features," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., Dec. 2018, pp. 6639–6649.

[32] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proceedings of the 31st International Conference on Neural Information Processing*

Systems, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., Dec. 2017, pp. 4768–4777.

[33] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: Association for Computing Machinery, Aug. 2015, pp. 123–137.

[34] J. Wilkes, "Yet more Google compute cluster trace data," Google research blog, Mountain View, CA, USA, Apr. 2020, posted at https://ai.googleblog.com/2020/04/yet-more-google-compute-cluster-trace.html.

[35] K. Wang, Y. Li, C. Wang, T. Jia, K. Chow, Y. Wen, Y. Dou, G. Xu, C. Hou, J. Yao, and L. Zhang, "Characterizing Job Microarchitectural Profiles at Scale: Dataset and Analysis," in *Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP '22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 1–11.

[36] ISAIC, "Powering the a.i.mbition of western canadian businesses," 2024, accessed: 2024-02-29. [Online]. Available: https://isaic.ca

[37] Amazon Web Services, "Cloud computing with aws," 2024, accessed: 2024-02-29. [Online]. Available: https://aws.amazon.com/what-is-aws/

[38] ——, "Amazon simple storage service (s3)," https://aws.amazon.com/s3/, 2024, accessed: 2024-03-15.

[39] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016, http://www.deeplearningbook.org.

[40] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning internal representations by error propagation*. Cambridge, MA, USA: MIT Press, 1986, p. 318–362.

[41] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[42] NVIDIA Corporation, "Nvidia tesla t4 gpu," https://www.nvidia.com/en-us/data-center/tesla-t4/, 2020, technical specifications and details available from NVIDIA.

[43] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Comput. Surv.*, vol. 53, no. 2, mar 2020.

[44] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[45] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[46] PyTorch Development Team, "Elastic run — pytorch documentation," https://pytorch.org/docs/stable/elastic/run.html, 2024, accessed: 2024-04-11.

[47] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[48] Apache Software Foundation, "Hadoop," accessed: 2024-03-20. [Online]. Available: https://hadoop.apache.org

[49] ——, "Apache spark," https://spark.apache.org/, 2020, accessed: 2024-03-15.

[50] "Amazon EMR - Big Data Platform - Amazon Web Services," accessed: 2024-02-29. [Online]. Available: https://aws.amazon.com/emr/

[51] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, May 2010, pp. 1 –10.

[52] V. K. Vavilapalli *et al.*, "Apache hadoop yarn: yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: Association for Computing Machinery, 2013.

[53] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[54] Apache Software Foundation, "Apache mllib," https://spark.apache.org/mllib/, 2020, accessed: 2024-04-15.

[55] ——, "Apache graphx," https://spark.apache.org/graphx/, 2020, accessed: 2024-04-15.

[56] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," *Proc. VLDB Endow.*, vol. 5, no. 7, p. 622–633, mar 2012.

[57] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[58] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford Digital Library Technologies Project, Tech. Rep., 1998. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.1768

[59] B. Rozemberczki and R. Sarkar, "Twitch gamers: a dataset for evaluating proximity preserving and structural role-based node embeddings," 2021.

[60] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[61] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[62] O. C. Jean-Baptiste Tien, joycenv, "Display advertising challenge," 2014, accessed: 2024-03-15. [Online]. Available: https://kaggle.com/competitions/criteo-display-ad-challenge

[63] "Amazon EC2," accessed: 2024-03-20. [Online]. Available: https://aws.amazon.com/pm/ec2/