GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Bachelor's Thesis

submitted in partial fulfillment of the
requirements for the course "Applied Computer Science"

A Quantitative and Qualitative Comparison of Machine Learning Inference Frameworks

Egi Brako

MatrNr: 21907203

First Supervisor: Prof. Dr. Julian Kunkel
Second Supervisor: Dr. Sven Bingert

Georg-August-Universität Göttingen
Institute of Computer Science

ISSN: 1612-6793

July 5, 2024

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 www.informatik.uni-goettingen.de

Abstract

As Artificial Intelligence (AI) continues to advance and impact diverse fields, ensuring universal access to its abilities becomes increasingly crucial. To access various AI models, they must be deployed to process inference requests. We conducted qualitative and quantitative analyses of popular open-source serving frameworks by evaluating their performance on three common Machine Learning tasks. This research aims to shed more light on the frameworks' respective strengths and weaknesses, consequently addressing the challenges posed by the process of selecting a method of serving the models. The qualitative comparison is carried out by taking into account the subjective characteristics of each framework and scoring them on a number scale. We then use Locust to run load-tests on these frameworks, log their quantitative results, and compare them with each other. Our results find that PyTorch TorchServe is the overall best-performing framework, consistently surpassing the other two in our performance test. We also found that some platforms had significant issues handling more complex models, showing incapabilities for handling specific Machine Learning tasks. Our findings show significant differences among the frameworks, contributing valuable insights for developers and researchers in selecting the most suitable framework serving Machine Learning models.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☑ To write individual passages, altogether to the extent of 5-10% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☑ Further, namely: making the graphs in the Results section prettier

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Acknowledgements

I would like to thank my supervisors, Prof. Dr. Julian Kunkel and Jonathan Decker, for answering any and all questions that I had. Without their help and invaluable advice this thesis would not have been possible.

Additionally, I'd like to thank the GWDG, for providing me with access to their computing cluster in order to carry out my experiments. Finally, I want to thank my family, my girlfriend, and my friends for being very patient and supportive with me during this time.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

AI Artificial Intelligence

AIaaS AI as a Service

API Application Programming Interface

ASR Automatic Speech Recognition

AWS Amazon Web Services

BART Bidirectional Auto-Regressive Transformers

CNN Convolutional Neural Network

CPU Central Processing Unit

DL Deep Learning

DNN Deep Neural Network

GPU Graphical Processing Unit

gRPC gRPC Remote Procedure Calls

HPC High-Performance Computing

HTTP Hypertext Transfer Protocol

IoT Internet of Things

JIT Just In Time

JSON JavaScript Object Notation

ML Machine Learning

NLP Natural Language Processing

ONNX Open Neural Network Exchange

OS Operating System

REST Representational State Transfer

RPS Requests per Second

SIF Singularity Image Format

XLA Accelerated Linear Algebra

1 Introduction

In this section, we will introduce the purpose and scope of the thesis, firstly in Section 1.1, by outlining the motivation behind comparing Machine Learning (ML) inference frameworks. We then detail the primary goals of the research in Section 1.2. In Section 1.3 we present the contributions made by this work to the field. Finally, in Section 1.4 we give a short outline of the thesis structure, providing an overview of what to expect in the following sections.

1.1 Motivation

In recent years ML has been acknowledged as a pivotal technology in many different domains, revolutionizing areas such as transportation [BH22], healthcare [Jav+22], and finance [DHB20]. With increasing reliance on ML models to make important decisions as well as help us in everyday life, the need to offer these models to a larger, broader audience is rising. A critical aspect of offering these ML solutions to users is serving the specific models. Serving is the process of deploying, maintaining, and managing ML models in production environments. This is normally accomplished with the help of serving frameworks (also referred to as inference frameworks). These frameworks ensure that the models are accessible, responsive, and capable of delivering accurate predictions or results to end-users in real-time.

Despite (or perhaps due to) the advancement in technologies in the field, selecting the appropriate serving framework remains a significant challenge. There are many inference frameworks, each with its own characteristics for serving ML models, various techniques, and addressing different needs. The performance of these frameworks can substantially impact the overall effectiveness of deployed models. While several inference frameworks exist, there is a gap in comprehensive comparisons that evaluate their performance and usability in different scenarios. Understanding which framework performs best under various conditions is paramount for developers and researchers aiming to optimize their ML workflows.

Our research addresses this research gap by providing a quantitative and qualitative comparison between three ML serving frameworks, namely: NVIDIA Triton Inference Server,[1] Pytorch TorchServe,[2] and TensorFlow Serving.[3] By assessing both performance metrics as well as usability factors, this study aims to identify the most suitable frameworks for diverse use cases. This will not only help in choosing the right framework based on the specific use case, but also motivate additional work into further development of the frameworks, to meet evolving requirements in the field of ML inference.

1.2 Goals

The overarching goal of this thesis is to understand how these frameworks compare to each other from a performance and usability point of view. Our research will be based on

[1] *Triton Inference Server* `URL:` https://developer.nvidia.com/triton-inference-server (visited on 02/07/2024)

[2] *TorchServe.* `URL:` https://pytorch.org/serve/ (visited on 02/07/2024)

[3] *Serving Models | TFX | TensorFlow* `URL:` https://www.tensorflow.org/tfx/guide/serving (visited on 13/06/2024)

the metrics chosen for both the quantitative and qualitative studies. Our methodology is systematic, ensuring that our results are as accurate as possible and reproducible.

The metrics for the performance evaluation are the latency and the throughput of each individual serving framework. By choosing latency as one of our metrics, we will be able to see how quickly each individual request to the serving platforms gets processed. Throughput, on the other hand, indicates the number of requests a framework can handle within a given timeframe, reflecting its capacity to manage high workloads. These will be measured through load-testing experiments with different virtual users, and showcase which of the frameworks perform the best. By varying the number of virtual users for the experiments, we can identify which framework delivers the best performance under various conditions, providing clear insights into their operational efficiency.

Usability is of equal importance because it determines how practical the frameworks are in real-world scenarios. High usability reduces the learning curve, minimizes development time, and helps avoid problems that could arise from poorly designed interfaces or insufficient documentation. We have chosen to evaluate the frameworks' usability by creating five metrics: user-friendliness, documentation quality, framework features, community support, and maintenance and update frequency. These are all useful in their own ways, but together they contribute to identifying a good framework. Metrics like user-friendliness and framework features are of the greatest importance. In the long run, a user-friendly framework contributes to higher productivity, better maintainability, and an easier development experience. All these metrics will help us understand which of the serving frameworks have the best usability.

By addressing these specific goals, this thesis will determine the most suitable framework for different ML tasks. This will help in understanding the strengths and weaknesses of each framework in practical scenarios.

1.3 Contribution

The contributions of this thesis are as follows

- Development of performance and usability metrics to assess the usefulness of ML inference frameworks.
- Results of experiments with regard to the developed metrics.
- Qualitative and quantitative comparison of three of the most popular ML inference frameworks.
- Report on the current state of ML inference frameworks.

1.4 Outline

This thesis begins with the introduction of common terms and concepts relevant to this research in Section 2. Then, in Section 3 we explore previous works. We go through papers and research, and talk about their relevance, as well as their advancements to the field. Section 4 elaborates on our methodology, also outlining the steps taken in preparing the frameworks and the experiments. In Section 5 we show the results of our experiments and clarify which of the frameworks perform the best. Later on, in Section 6, we discuss the implications of our results, as well as any substantial problems encountered during

our experimental set up and execution. Finally, in Section 7 we summarize our research and talk about future work in this field.

Section Summary In this section we have set the stage for our comparative study on ML inference frameworks by outlining the motivation, objectives, and scope of the research. The main goal is to evaluate the performance and usability of TorchServe, TensorFlow Serving, and Triton Inference Server in handling various ML models. Understanding these differences is paramount for researchers and developers in selecting the best framework for their specific needs.

2 Background

Here, we will discuss the relevant technologies to this thesis. Firstly, in Section 2.1 we offer a deeper understanding of the serving frameworks' designs, configurations, specific functions, and the unique features that played a part in our selection of them. In Section 2.2 we explain High Performance Computing, its differences and advantages to normal computing. Section 2.3 goes into the concept of containerization and Singularity, our containerization platform of choice for this research. Finally, Section 2.4 discusses the concepts of load-testing and some of its well-known metrics, as well as our chosen solution for this task, Locust.

2.1 Model Serving Frameworks

In essence, serving frameworks are software platforms that enable ML models to take in inputs and return outputs, acting as a bridge between the model and real-world data. These frameworks are important in handling multiple requests and making good utilization of computational resources.

2.1.1 TensorFlow Serving

TensorFlow Serving[4] is a system designed for the deployment of ML models in practical applications. First introduced in the research paper from Google [Ols+17], it is part of the TFX[5] ecosystem and is optimized for serving TensorFlow models. It allows the integration of these models into production environments by providing an extensive Application Programming Interface (API) for other software and users to access the models' capabilities. It supports API calls through both Hypertext Transfer Protocol (HTTP)/Representational State Transfer (REST) and gRPC Remote Procedure Calls (gRPC) protocols. TensorFlow Serving is able to handle multiple requests concurrently and offering the possibility of having the inputs batched. It also enables uninterrupted updates of models without downtime to the system, allowing for continuous availability.

TensorFlow Serving's architecture revolves around **Servables** (the model abstraction), which "are the underlying objects that clients use to perform computation (e.g., a lookup or inference)"[6]. These are managed by **Loaders** that handle their lifecycle, including loading and unloading into memory. The **Manager** handles the full lifecycle of Servables, including loading, serving, and unloading Servables, as well as tracking all Servables' versions. **Clients** interact via APIs to request servables, with TensorFlow Serving dynamically managing versions to allow seamless updates and high-performance inference across different deployment scenarios.

Another important part of TensorFlow's architecture is SavedModel Signatures.[7] Signatures are key components that define how a model's functions can be accessed and

[4] *GitHub - tensorflow/serving* `URL:` https://github.com/tensorflow/serving/ (visited on 20/06/2024)

[5] *TensorFlow Extended (TFX).* `URL:` https://github.com/TensorFlow/tfx (visited on 13/06/2024)

[6] *TensorFlow Serving Architecture.* `URL:` https://www.TensorFlow.org/tfx/serving/architecture (visited on 13/06/2024)

[7] *A Tour of SavedModel Signatures.* `URL:` https://blog.tensorflow.org/2021/03/a-tour-of-savedmodel-signatures.html (visited on 18/06/2024)

employed after deployment. Every SavedModel has a default signature, or a custom one can be created. These serving signatures also constrain the allowed input sizes, ensuring models work correctly across different applications without needing the original model code.

2.1.2 PyTorch TorchServe

TorchServe[8], developed by PyTorch, is a framework designed for deploying multiple ML models efficiently. Its architecture consists of two main components: the frontend and the backend. The frontend, implemented in Java, manages incoming requests from clients, takes care of dynamic scaling, and provides APIs for inference requests. Key components here include an **API gateway** for HTTP/gRPC request handling and response aggregation, and a **model lifecycle manager**, responsible for handling model lifecycles including loading and unloading. On the backend, written in Python[9], the **Model Server** component initializes the framework and loads configurations, whereas the **Model Loader** dynamically loads models from storage, including the model's handler, which carries out preprocessing and postprocessing of the input data. These components, in tandem, enable TorchServe to scale models, integrate custom functionalities through plugins, and manage resources efficiently.

Another important feature of TorchServe is **inference handlers**. The model applies the handlers to preprocess the input data, customize the model invocation for inference (if necessary), and post-process the model output before sending back a response. They work with two main types of information: 'data' from the incoming request, and 'context' from the TorchServe environment, which includes such things as model name, model directory, and Graphical Processing Unit (GPU) (or Central Processing Unit (CPU)) details. TorchServe provides default handlers[10] for common tasks (e.g. image classification or object detection). Typically, users only need to override preprocess or post-process methods in the BaseHandler, which highlights the flexibility PyTorch provides in creating custom handlers.

2.1.3 NVIDIA Triton Inference Server

NVIDIA Triton Inference Server[11] is a powerful, flexible framework for deploying ML models in production environments. Its architecture revolves around a **client** API supporting both HTTP/REST and gRPC endpoints, while the **server** manages model loading, and optimization techniques, like dynamic batching, or TensorRT optimization. Triton can run inference on both CPUs and GPUs, including NVIDIA accelerators which enable it to deliver low latency and high throughput. Models are stored in the repository and loaded into memory upon server startup, with each model being managed as a separate instance by Triton's backend handlers. The client takes care of the necessary pre-and post-processing steps for the inference data, packaging it in a format that the Inference Server can understand. It then sends this data to the server to perform the inference, and afterward, collects the resulting output. This modular approach enables efficient model

[8] *GitHub - pytorch/serve* `URL:` https://github.com/PyTorch/serve (visited on 13/06/2024)

[9] *Python.* `URL:` https://www.python.org/about/ (visited on 30/06/2024)

[10] *TorchServe default inference handlers.* `URL:` https://PyTorch.org/serve/default_handlers.html (visited on 30/06/2024)

[11] *GitHub - triton-inference-server/server* `URL:` https://github.com/triton-inference-server/server/ (visited on 20/06/2024)

loading and inference processing, thus optimizing performance and resource utilization. It enables models to be deployed both on-cloud and on-premises, making it flexible for any requirement.

What sets Triton apart from other serving frameworks is its multi-framework support. It serves models from most ML frameworks, including TensorFlow, PyTorch, and ONNX, within a single server instance. This flexibility gets rid of the need for multiple serving systems, making it simpler to deploy and manage the workflow. Another big advantage of Triton Inference Server is that it can be deployed in containerized environments such as Docker and Kubernetes, making the whole setup process much easier. By utilizing this, Triton can be set up in a dockerized environment, pull in models, and let it handle the optimization of the models for the best performance.

2.1.4 Summary

Although we gave a comprehensive introduction of each serving framework above, we could not cover every detail. Below, Table 1 showcases some other differences and similarities between our chosen serving frameworks.

Frameworks	PyTorch Torchserve	TensorFlow Serving	NVIDIA Triton
Model Format	TorchScript, ONNX	TensorFlow SavedModel	ONNX, TensorFlow SavedModel, TorchScript, etc.
Release Date	April 2020	December 2015	March 2019
Containerization	Yes	Yes	Yes
Multiple Model Versions	No	Yes	Yes
Input Data Format	Binary (raw bytes)	JSON	Declared in config file
GPU support	Yes	Yes	Yes

Table 1: Characteristics of the serving frameworks

2.2 High Performance Computing

High-Performance Computing (HPC) is a class of applications and workloads that solve computationally intensive tasks[12]. These tasks often surpass the capacity of conventional hardware due to their intensive computational requirements or unusually large data volumes. The architecture of HPC systems supports both intra-node and inter-node parallelization, allowing it to harness the power of multiple machines to solve intricate problems. HPC proves invaluable in a range of areas, such as finance, medicine, engineering, scientific computing, and more, where its high-throughput, low-latency capabilities can provide solutions that might otherwise be unattainable.

Beyond their raw computational power, HPC systems are characterized by their advanced hardware components and network infrastructures, offering high-speed data transfers crucial for distributing tasks across compute nodes effectively. HPC systems often incorporate GPUs to enhance their parallel computing capabilities, making them highly

[12] *What Is High Performance Computing - Intel* `URL:` https://www.intel.com/content/www/us/en/high-performance-computing/what-is-hpc.html (visited on 14/06/2024)

effective for compute-intensive tasks such as Deep Learning (DL). With direct access to state-of-the-art processing architectures, accelerators, and optimized libraries, HPC applications are designed to take full advantage of the underlying hardware capabilities.

2.3 Containerization and SingularityCE

Containerization is a method of packaging applications with all their necessary components, such as libraries and dependencies, into isolated units called containers. These containers ensure that the applications run the same way in any environment. Unlike virtualization, which creates a full operating system, containers share and leverage the host system's Operating System (OS), making them more efficient and lightweight. Using the same OS simplifies the deployment and management of applications, allowing them to be easily moved from one system to another, or between different stages of development and production without any compatibility issues. Containerization enables the consistency, reproducibility, portability, as well as the security of software.

Singularity Community Edition[13] is an open-source containerization platform, "created to run complex applications on HPC clusters in a simple, portable, and reproducible way". Unlike other containerization tools (e.g. Docker[14]), Singularity is built to appease the unique needs of scientific research communities. This means that Singularity containers do not require administrative privileges to run, making it more secure for researchers who work in a shared-resource environment. Singularity focuses on integration over isolation by default, leveraging its design strategy to seamlessly connect with the host system's resources, such as GPUs, high-speed networks, or parallel filesystems, rather than keeping them isolated. All Singularity containers are contained in a single Singularity Image Format (SIF) file, making the containers easy to transport and share.

2.4 Load Testing and Locust

Load-testing is a process which evaluates the performance of a system under a specific load. It involves simulating multiple virtual users accessing the system simultaneously to determine how it behaves under stress. This helps identify potential bottlenecks and performance issues, and ensures that the system can handle expected user traffic. Load-testing is crucial for verifying the reliability, stability, and scalability of applications before they are deployed in a production environment, ensuring they can maintain optimal performance during peak usage times.

Load-testing can measure certain metrics when carried out. Specifically in our research, we're interested in two: prediction latency and throughput. Prediction latency is "the time it takes to render a prediction given a query"[Cra+17]. It indicates the speed at which individual requests are processed. Lower latency is desirable as it signifies faster responsiveness to user actions. Throughput, on the other hand, "refers to the number of classifications made per time quanta" [Cho18]. It represents the system's capacity to handle concurrent user interactions effectively. A higher throughput value is wanted as it implies the system can process a larger number of requests simultaneously. Both latency and throughput are crucial indicators of performance during load testing.

When focusing solely on latency and throughput in a straightforward manner, the two

[13] *SingularityCE | Sylabs* URL: https://sylabs.io/singularity/ (visited on 14/06/2024)
[14] *Docker.* URL: https://www.docker.com/ (visited on 14/06/2024)

metrics have an inverse relationship, usually meaning the lower the prediction latency, the higher the throughput. This is not always true though, especially when we consider such concepts as batching, which groups multiple requests in one batch, meaning they get processed at the same time. Batching can improve throughput by reducing the overhead needed for processing individual requests. However, it may increase latency for individual requests within the batch due to the waiting time until the entire batch is processed.

Our chosen load-testing solution, Locust,[15] is an open-source tool designed to help developers assess the performance of their applications. It allows users to create custom test scenarios in Python code, simulating millions of simultaneous virtual users to evaluate how the system performs under heavy load. Locust also provides a web-based interface, which offers real-time monitoring, as well as detailed reports and graphs, making it easy to identify performance issues and bottlenecks. Locust can be utilized to test any system by simply wrapping calls to REST APIs, or implementing bespoke load patterns to meet specific testing needs. This adaptability makes Locust very "hackable", allowing it to be an effective tool for ensuring that applications can handle high traffic and perform reliably when they are deployed in production environments.

Section Summary In this section we have provided an overview of the key concepts and technologies underlying ML inference frameworks. We have discussed the architecture and functionalities of all relevant solutions, focusing on why they are useful to us. This foundational knowledge is essential for understanding the comparative analysis that follows.

[15] *What is Locust.* `URL:` https://docs.locust.io/en/stable/what-is-locust.html (visited on 30/06/2024)

3 Related Work

As the field of ML grows, the focus extends beyond individual models and their training to include the efficient deployment and serving of these models to users. In this section, we review the relevant literature to our own research. In Section 3.1 we look at research that also employs open-source frameworks on ML serving. Section 3.2 gives an overview on previous work where researchers have created their own serving frameworks, and compared these with established solutions. Finally, in Section 3.3 we look at research in the field of edge-computing. While significant work has been done on ML inference, the specific area of serving platforms meant for general use has received less attention.

3.1 General inference analysis

Works such as [Red+19] offer a great approach to performance assessment, establishing new benchmarks for measuring the efficiency of ML inference, covering different aspects of ML like vision, language, and Natural Language Processing (NLP). They focus on three main aspects, namely: performance metrics, accuracy/performance tradeoff, and evaluation of AI inference accelerators. Although they make great strides for benchmarking ML models, some aspects are not looked at. Due to limited resources and time constraints, they were only able to look at a few models, and were forced to leave out models like BERT [Dev+18] and transformers [Vas+17]. The authors were also unable to include more applications such as speech recognition or recommendation.

Following up on the original MLPerf paper, the authors of 'The Vision Behind MLPerf: Understanding AI Inference Performance' [Red+21] extend the benchmarking suite to cover a wider range of hardware and software systems, thus enhancing the scope of these performance evaluations. This paper improves in defining evaluation scenarios which, like our research, better reflect practical deployment environments. It also introduces a modular design, ensuring the benchmarks can evolve with the rapidly changing landscape of ML applications. Despite these advancements, there are still areas for improvement. The ML hardware discussed in this paper is created and tested in a vacuum, without taking into account the complete application setting. Successful AI applications heavily rely on preprocessing and postprocessing stages, storage, and communication systems, all of which must be factored in when evaluating the overall performance of AI systems. Furthermore, this paper does not carry out a concrete comparison of serving frameworks intended for general use.

The closest work to our research is the one done by researchers from the Biano-based study [AI21]. They focus specifically on the effectiveness of different neural network serving platforms intended for general use. They examined three prominent serving platforms- TensorFlow Serving, PyTorch TorchServe, and NVIDIA Triton Inference Server- to analyze the efficiency and reliability of these platforms in deploying Deep Neural Network (DNN) models for production. However, it also has its limitations. The study only focuses on an image classification task and leaves out many others, such as speech recognition or text processing. Furthermore, the platforms being tested have continuously evolved in the past three years, changing a lot since this research was carried out, meaning the results obtained are bound to be outdated. Regardless, the study offers valuable data on the speed, response times, and reliability of these platforms.

3.2 Inference on custom serving systems

Further work on comparisons of serving frameworks has been done by authors creating their own serving solutions, and then comparing them with state-of-the-art existing solutions. Chard et al. [Cha+19] present a platform called DLHub, designed to address the specific needs of ML by offering scalable, low-latency model serving capabilities. Comparative tests demonstrate DLHub's higher performance, especially taking advantage of techniques like memoization and batching, against other serving frameworks like TensorFlow Serving or SageMaker[16]. The research paper, however, does not compare all the available open-source frameworks available. Its performance analysis is also limited, focusing only on small models within a single ML task.

Similar work comes by Crankshaw et al. *Clipper: A Low-Latency online prediction serving system* [Cra+17] introduces a robust system for real-time ML predictions, emphasizing its modular architecture designed to enhance model deployment, accuracy, and performance. Clipper, like DLHub, implements various techniques such as caching, adaptive batching, and model composition, which optimize prediction latency and throughput. It achieves performance comparable to TensorFlow Serving while offering functionalities like dynamic model selection. Clipper's evaluation primarily focuses on specific benchmarks, while broader, real-world applicability is not explored that thoroughly. The paper is no longer being maintained, which means the data it has gathered is very outdated, seeing as its counterpart, TensorFlow Serving, is continuously updated. Even so, this paper is important as it provides foundational insights, techniques, and ideas that influence current research and development in ML serving systems.

3.3 Inference on edge devices

DL is extensively applied in various fields like computer vision, NLP, autonomous driving, and many others. However, processing data on end devices like smartphones and Internet of Things (IoT) sensors requires a lot of computational resources [Shi+16]. Edge computing, where a fine mesh of compute nodes are placed close to end devices, is incredibly valuable for DL because it brings processing power closer to where data is generated. This approach not only meets the demand for high throughput and low inference latency, but also offers advantages in privacy, bandwidth efficiency, and scalability [Shi+16]. There has already been a lot of research into bringing DNN computation to edge devices for numerous tasks, such as computer vision [DGN17] [Ran+18], or NLP [Jia+20].

Furthermore, the scientific community has been researching and carrying out performance analysis on ML inference on edge-class devices. Most notably, Chen et al. [CR19] provides a detailed review of the latest developments in DL and edge computing. It covers how DL is applied at the network edge, methods for quickly running DL tasks on end devices, edge servers, and the cloud, and how to train models across multiple edge devices. It also discusses ongoing challenges related to system performance, network management, benchmarks, and privacy. Although it gives an extensive, comprehensive overview of DL inference, its scope is focuses on edge devices, leaving out broader inference frameworks designed for general use.

[16] *Amazon SageMaker*. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html (visited on 10/06/2024)

Section Summary In this section we have gone over existing literature on the performance of ML inference frameworks. The review has shown that while significant work has been done, there is still a need for a comprehensive, up-to-date comparison incorporating the latest versions of these frameworks. By examining past studies and their methodologies, we have identified gaps in the field, as well as opportunities for our research.

4 Methodology

This section outlines the methodology for evaluating the inference frameworks. We start Section 4.1 by formally presenting our research questions. Section 4.2 and 4.3 dictate our formal performance and usability metrics, respectively. In Section 4.4 we outline the methods of our experiments, detailing the specific ML tasks and models chosen for our experiments. Section 4.5 descripes the setup of our experiments, discussing our configurations to ensure replicability and accuracy in the results. In Section 4.6 we detail how our experiments were executed, along with the hardware and software choices. Finally, Section 4.7 summarizes our methodology, and shows why it is systematic and research-worthy.

4.1 Research Question

Our aim in this research is to perform both a qualitative and quantitative analysis of the aforementioned ML frameworks. The challenge here lays in merging these aspects to form a comprehensive research question. The common part that stands out is this broad concept of "usefulness". Both performance metrics and the practical, user-oriented aspects contribute to determining how useful these frameworks are, under various conditions, for different users. Thus, we can consider the general research question as containing two parts:

RQ1: Which of these frameworks have the highest performance? Here, we analyse the quantitative performance of the ML inference frameworks.

RQ2: Which of these frameworks have the best usability? Here we will analyse the qualitative characteristics of these frameworks.

Because we are only talking about the serving platforms of each framework, we will ignore the comparison of training, or fine-tuning methods on the respective frameworks, and assume that the official implementations in the model zoos of the respective frameworks are the exact models that are required in our scenario. We wanted to choose a (real-life) use case and a testing method that answered the research questions as best as possible. We choose to look at two use cases:

UC1: A user with limited technical skills looking for a straightforward solution for model deployment, without delving into intricate details. The concept of usefulness would be most appreciated in this use case. The user can select one of the frameworks, pick a respective model that best suits their task, and serve it without encountering setbacks.

UC2: A user planning to incorporate AI models into a broader model serving infrastructure, offering AI as a Service (AIaaS). The most important aspect here would be the platforms' performance in terms of speed and resource utilization and how these scale with the simultaneous handling of multiple models. Secondary aspects here would be things such as the features and customization of the serving frameworks.

4.2 Performance

In order to answer **RQ1**, we would need to define formal methods and metrics to find out the best-performing framework. For this, we chose two different methods of load-

testing (scenarios) for each serving framework: multi-stream load test, and single-stream throughput test. The former works by regularly querying the serving platform, and firing the next request as soon as the previous one completes. This, coupled with a variable amount of virtual users, are going to keep the platform under constant heavy load, thus simulating a worst-case scenario. Although most systems will never be under constant load for long periods of time, simulating a worst-case scenario helps ensure the system is reliable, even under heavily demanding circumstances.

The single-stream test measures how well the platform handles a continuous flow of requests, one at a time, as quickly as possible. It helps determine the maximum rate at which the platform can process inference requests without any delays or slowdowns. This is simulated by utilizing Locust's event-based approach. We test the platform with only one virtual user, waiting until the request is processed, before sending another one.

After defining the methods, we need to define the metrics that we can evaluate performance with. In both of these scenarios, three different metrics will be measured, namely: Throughput (in Requests per Second (RPS)), Latency (in milliseconds needed per response), and Failures (in failures per second). These measurements helped us understand how well the systems work in different situations. By evaluating these metrics, we provide a comprehensive analysis of how well various ML serving frameworks can handle high demand and deliver fast responses. This directly supports the needs of **UC2**, ensuring that the chosen framework will maintain high performance, reliability, and responsiveness, which are essential for an AIaaS platform's success.

4.3 Usability

A usability assessment is difficult to conduct without set criteria. In order to answer **RQ1**, we must define formal metrics, and then strictly evaluate the serving frameworks. Our usability analysis focuses on how effortlessly a user can set up and serve the chosen models, along with the complexity of the serving process. Given that all of the frameworks under review are considerably sized projects, it is reasonable to expect a high level of support for users, both in the documentation and through the community. It is difficult to judge qualitative characteristics in a specific way, therefore we have written the following loose criteria, which will serve as our reference parameters throughout our analysis.

1. User-Friendliness:
    - How difficult was it to set up and deploy the framework?
2. Documentation Quality:
    - Does the documentation assist with common issues?
    - Is the documentation kept up-to-date?
    - Is the documentation understandable by new users?
3. Project features:
    - Can the framework accommodate and serve models generated from various popular ML libraries?
    - Is it possible to define a custom preprocessing or postprocessing step?

4. Community support:
    - How many active users or contributors does the project have on GitHub?
    - What is the usual response time for an issue resolution?
    - Are online forum discussions or help resources widely available?
5. Maintenance and Update Frequency:
    - What is the regular release cycle for updates and major releases?

The criteria listed above will be used to make a usability comparison between the different frameworks. The main points will all receive a score ranging from one to five. Through this, a systematic comparison can be made based on the observations and findings made throughout this research project. By conducting such a detailed analysis, we can present a comparison that highlights the strengths and weaknesses of each framework. Not only this, but we can directly apply this analysis to the needs of **UC1**. For this use case, the primary concern is the user-friendliness of the serving framework. Key factors such as the availability of understandable documentation, or helpful framework features are crucial for a user needing a quick and easy experience with the framework. Thus, the comprehensive usability analysis directly informs which framework would best suit a user focused on minimizing setup complexity and maximizing support and ease of use.

4.4 Machine Learning Tasks

When choosing the models, we refrained from changing too much in the configurations of the specific serving platforms to ensure our methodology remains true to the use case scenario introduced earlier. As we will see further on in this thesis, there were certain exceptions to this, where it was necessary to change configurations of some frameworks for the serving to be possible at all. When choosing the ML tasks, we focused on ones that are relevant to the field, and are different in nature from each other. We decided on three important ones, namely: Image Classification, Automatic Speech Recognition (ASR), and Text Summarization.

4.4.1 Image Classification

Image classification is the task of analyzing a picture and automatically identifying and labeling the objects or subjects it contains. It is applied in many areas like healthcare, self-driving cars, and social media. There are many well-known benchmarks and datasets in this field that allow us to compare performance. These benchmarks provide widely accepted metrics, making it easier to compare the different models.

For this task we chose the ResNet50 [He+15] model. It is a variant of a Convolutional Neural Network (CNN), explicitly designed for image classification tasks. It contains 50 layers with Residual connections. The architecture is well-suited for processing image data and extracting hierarchical visual features. Due to its popularity in both research and industry, ResNet50 serves as a dependable standard for assessing the effectiveness of new algorithms or techniques in image classification.

4.4.2 Automatic Speech Recognition

ASR is the task of transforming a spoken language into written text. This involves analyzing the sound waves in a voice file and transcribing them to text. ASR can be applied to a variety of tasks such as speech-to-text, controlling software by voice command, and aiding language translation systems. Voice recognition models have a different architecture from image classification models. They deal with high-dimensional data and require continuous learning to adapt to different accents and speech patterns.

For this task we have chosen the Wav2Vec2 [Bae+20] model, specifically Wav2Vec2-base-960h, which is a model pre-trained and fine-tuned on 16kHz sampled speech audio taken from the LibriSpeech [Pan+15] dataset. It is a state-of-the-art model, designed for ASR. It outperforms other speech recognition models on several benchmarks. Wav2Vec2 is a good choice due to its self-supervised learning being able to directly find useful representations from raw audio inputs and provides a fair ground to investigate different ML inference frameworks' capabilities without the influence of feature extraction techniques. Comparatively, other models might require extensive preprocessing or feature extraction techniques, which adds complexity and may introduce biases.

4.4.3 Text Summarization

Text summarization is a task in NLP that involves shortening a text in a way that captures the main points or themes of the original text. It processes large amounts of textual information quickly, helping users to understand the content without having to read it in its entirety. It is a sequence-to-sequence task, which can shed light on how different frameworks handle and perform with high-dimensional, textual data.

The Bidirectional Auto-Regressive Transformers (BART) [Lew+19] model is a sequence-to-sequence model developed by Facebook AI, and the model we have chosen for this task. It is designed for several NLP tasks, such as text generation, translation, and summarization. BART functions by first corrupting the text and then learning to reconstruct it, working bi-directionally. The "-large-CNN" variation is specifically trained on article and summary pairs from the CNN and Daily Mail dataset, first introduced in the work by Hermann et al.[Her+15]. This model tackles the challenges of language understanding, context preservation, and summarization in the broad field of NLP.

4.5 Setup

In this subsection, we describe the hardware and software configurations that ensure the replicability of our results. All the model implementations are taken as-is, directly from the respective model zoos, such as the Torchvision and the Keras packages. In some cases, the pre-trained model weights were not available in the model zoos, in which case we took their official implementations in Huggingface. Both PyTorch and TensorFlow have different formats for the pre-trained model weights, meaning they had to be saved separately. Usefully, NVIDIA Triton Inference Server serves models from both these frameworks and accepts any format of model weights that are accepted by the individual frameworks.

```python
import torch
from torchvision.models.resnet import resnet50, ResNet50_Weights

model = resnet50(weights=ResNet50_Weights.DEFAULT).to("cuda:0")
model.eval()
example_input = torch.randn(1,3,224,224).to("cuda:0")
traced_model = torch.jit.trace(model, example_input)
torch.jit.save(traced_model, "model.pt")
```

Listing 1: Exporting the (PyTorch) ResNet-50 model

The saving of the models was different for both frameworks. On the PyTorch model seen above, we take advantage of PyTorch's Just In Time (JIT) compiler to export the model and prepare it for serving. The chosen method is tracing. Tracing records operations performed on a given input, which is useful for straightforward models (such as ResNet-50). Saving of the TensorFlow was similar, also only requiring a few lines of code. The main difference is that, unless the model weights are available in the TensorFlow Hub (currently fully integrated with Kaggle)[17], the weights have to be downloaded manually. This, however, was not a factor in our research.

Locust was our load-testing platform of choice for multiple reasons. It offers distributed load generation, meaning that all the events and virtual users can scale to multiple processes, and even multiple processors. This is very important in our case since we do not want to overload the model serving platform by running hundreds of virtual users in the same processor, as this might lead to inconsistent data. We can take advantage of this since we have plenty of available resources in the HPC. Locust's event-based approach can handle a large number of users more efficiently, because it does not create a separate thread or process for each user (unlike other load-testing tools such as Apache JMeter[18]). Instead, it leverages asynchronous I/O operations and event loops to manage tasks, thus reducing resource usage and improving scalability. Finally, Locust is very small and flexible, which we take advantage of to write tests in pure Python. This simplified the entire process because apart from the SLURM files, the rest of our code is written in Python.

4.6 Execution

After choosing our serving frameworks, models, and load-testing method, we have to consider putting all of these together in systematic, reproducible experiments. Due to the fact that we are running all the tests in an HPC environment, we chose Singularity to containerize our frameworks. In every experiment, one container serves the model, and another runs our load-tests. All of the containers of the frameworks have their own configuration, which are easily reproducible due to the Singularity definition files. For TensorFlow and NVIDIA Triton Inference Server, we have created the definition files stemming from their official Docker Hub implementation, only binding extra files and folders as needed for serving. The PyTorch container, on the other hand, is a bit different. It was implemented from an official base Python container from Docker Hub, after which we installed the necessary packages to serve the respective models.

All experiments were conducted on a NVIDIA Quadro RTX 5000 GPU,[19]. They

[17] *Kaggle Models.* URL: https://www.kaggle.com/discussions/product-feedback/448425 (visited on 10/06/2024)

[18] *Apache JMeter.* URL: https://jmeter.apache.org/

[19] *NVIDIA Quadro RTX 5000.* URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-

include a consistent range of user counts: 1, 10, 25, 50, 75, and 100. Also, every test was run 5 times, and the average of all runs was taken. We chose this to increase the reliability and reproducibility of the results, minimizing the impact of outliers and background noise.

The load-tests were run on the HPC with pre-written SLURM batch files, so that we could always request and utilize the same resources. The testing approach is as follows:

1. Load Singularity container with model on the GPU
2. Run model warm-up tests
3. Run test with x number of users, log and save results to csv format
4. Redo identical test 5 times in total
5. Increase x until 100, go to step 2
6. Stop Singularity container

4.7 Method

Our chosen methodology is research-worthy for a few distinct reasons. Firstly, it is grounded in clearly defined research questions that guide the entire study. Section 4.1 presents two main research questions that focus on the performance and usability of different ML inference frameworks, guaranteeing that the study addresses both quantitative and qualitative aspects comprehensively .

The evaluation is systematic, and clearly structured around two main aspects: performance and usability. These aspects are broken down into measurable criteria, such as throughput, latency, and failure rates for performance, and user-friendliness, documentation quality, project features, community support, and maintenance frequency for usability. This comprehensive set of criteria ensures a thorough and balanced evaluation of the frameworks.

The ML tasks chosen in Section 4.4 represent a wide range of applications in the field of ML. By selecting different types of tasks, the research covers a broad spectrum of potential applications, making the findings more generalizable and relevant to various scenarios.

Throughout our experiments, we maintained the integrity of the evaluations. In order to ensure this, in Section 4.5 and 4.6 we show that the configurations of the serving frameworks were kept as consistent as possible. Not only this, but the experimental setup, including hardware and software configurations, is meticulously described to ensure that the results can be replicated. This approach minimizes variability due to configuration differences and ensures that the performance comparisons are fair and accurate.

Section Summary In this section we have described the experimental setup and procedures used to evaluate the chosen frameworks. This includes the selection of models, performance metrics, and testing environment. The consistency of our methodology ensures that the results are reliable and reproducible, allowing us to proceed with the upcoming analysis.

visualization/quadro-product-literature/quadro-rtx-5000-data-sheet-us-nvidia-704120-r4-web.pdf (visited on 17/06/2024)

5 Results

This section will showcase the results received from our experiments, as well as conduct some comparisons between the frameworks. Section 6 In Section 5.1 we show the performance results of our tests, going through each of the models and comparing them with their respective counterparts. Later, in Section 5.2 we talk about the usability results of our experiments. Finally, in Section 5.3, we give our evaluation on each of the serving frameworks, both concerning performance as well as usability.

5.1 Performance results

Our chosen method not only increases reproducibility of our results, but also provides more data for statistical analysis, allowing us to understand the range and standard deviation of the results, thus strengthening the conclusions drawn from the study.

We used tools such as `nvitop`[20] to observe the resource utilization of our HPC node. We noticed that the utilization of the CPU constantly remained around 30% for all models. This is actually a design choice, coming from the distributed load generation feature of Locust, which ensures balanced distribution of load across all worker nodes, preventing any single node from becoming a bottleneck. In addition, we noticed that the GPU utilization remained around 30-50% for smaller models such as ResNet50 and the truncated version of Wav2Vec2. Even though the BART and Wav2Vec2 models are quite large, the GPU usage never exceeded 90%, showing that all of the frameworks deal very well with increasing demand. All of these measures that we have taken prove that the results received from our experiments are consistent and replicable.

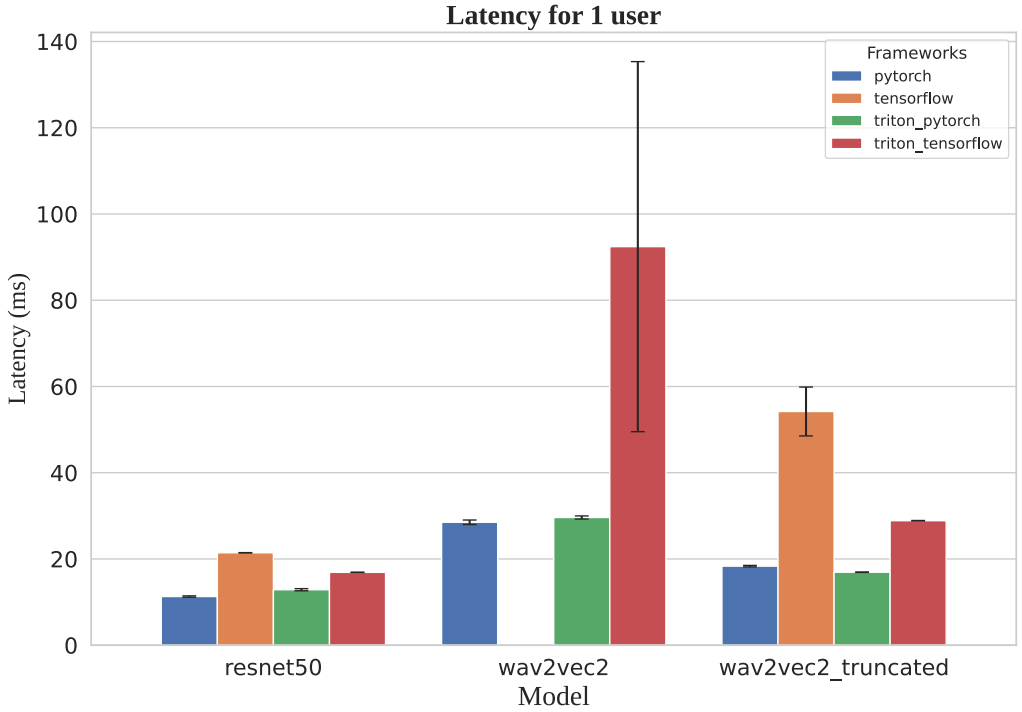

Figure 1: Latency results for 1 virtual user (in milliseconds)

[20] *Github - XuehaiPan/nvitop* `URL:` https://github.com/XuehaiPan/nvitop (visited on 03/07/2024)

As we show in Figures 1 (and Table 5 in the Appendix), the range of the results was small, with the only exception of this being the Wav2Vec2 model running on the Triton (TensorFlow) framework (as shown in Figure 1). This small range shows that our experimental results were consistent, suggesting that variations in the results can be attributed to actual performance differences rather than methodological inconsistencies or random errors.

In Figure 1 we can see the prediction latency for all the models when testing the framework performance with only one virtual user (our single-stream test). We have chosen to show only the graphics of the tests with 1 and 100 users (found in Appendix Figure 5), since the *prediction* latency is more or less similar for a specific model and framework combination, no matter the user load. The reason why the *value* of the latency increases when raising the user count is because all submitted requests must wait in the queue until all others before it have finished. Since the throughput stays constant, raising the number of incoming requests (user count) will raise the amount of time that a request waits in the queue. This phenomenon can easily be verified by taking a look at the internal logs of the serving frameworks, which show the precise inference time.

5.1.1 ResNet50

As we can see in Table 2, for ResNet50, PyTorch consistently outperformed other frameworks in terms of throughput across all user counts. In the test with one virtual user, PyTorch achieved the highest throughput of 79.79 RPS, while Triton (PyTorch) and Triton (TensorFlow) reached 74.26 and 57.33 RPS, respectively. And although PyTorch has the highest throughput, Figure 2 shows that Triton's (PyTorch) performance is trailing not too far behind. TensorFlow Serving is in this case definitively slower, reaching only 5.56 RPS.

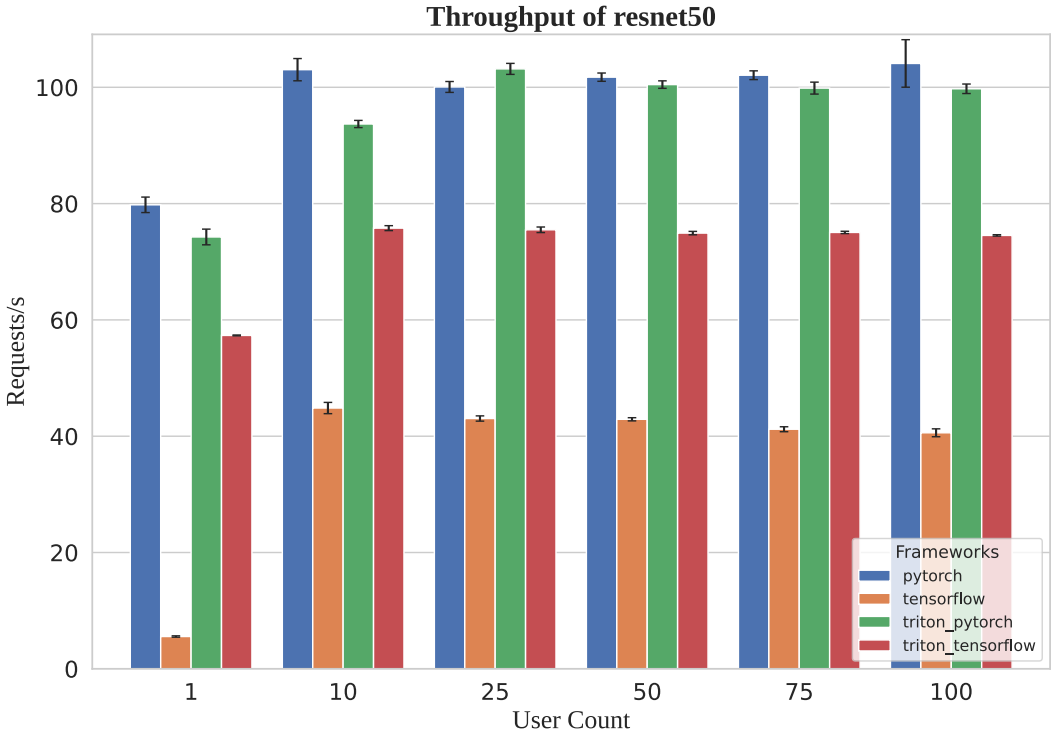

Figure 2: Throughput of the ResNet50 model (in requests per second)

User Count Framework	1	10	25	50	75	100
PyTorch	**79.79**	**103.03**	100.06	**101.74**	**102.08**	**104.10**
TensorFlow	5.56	44.85	43.04	42.91	41.20	40.59
Triton (PyTorch)	74.26	93.69	**103.17**	100.47	99.86	99.74
Triton (TensorFlow)	57.33	75.78	75.50	74.93	75.04	74.53

Table 2: Throughput of the ResNet50 model (in requests per second)

When considering all of the frameworks, the reported throughputs for the ResNet50 model are generally good. Apart from TensorFlow Serving, the fact that the other frameworks' throughputs are (consistently) around the 100 RPS mark is really good. It shows that they are able to handle a very large number of requests at the same time without any slowing down.

Similar to the throughput, Table 9 in the Appendix shows us that PyTorch exhibited a higher performance compared to the other frameworks when considering latency, with an average response time of 11.29 ms in the test with one virtual user, followed closely once again by Triton (PyTorch), at 12.87 ms. Triton (TensorFlow) had 16.89 ms, and TensorFlow showed the highest latency of all the frameworks, at 21.43 ms. The same relationship between the frameworks can be seen in Figure 5 in the test with 100 users, although the numbers change slightly in user counts 10-75.

These latency scores, although they differ from each other, show that the frameworks' general performance is excellent. With Pytorch and Triton, the average processing time for any request is (at worst) under one second, even when considering 100 concurrent users. TensorFlow and Triton (TensorFlow) also offer a worst-case response time of around 1.3 seconds. In a realistic scenario, even if the system were under the heaviest load, this would be a very short time to wait for a response. This can be attributed to the speed and efficiency of the frameworks, but also to the small size of the model, as we will see with the other models.

5.1.2 Wav2Vec2

Due to the imposed usability constraints, it was impossible for us to get the base Wav2Vec2 model running in TensorFlow. Therefore, the performance analysis for this model will not feature the TensorFlow Serving framework. When it comes to throughput, we can tell from Figure 4 and Table 10 that Triton and PyTorch displayed comparable performance at lower user counts, but Triton excelled as the load increased. In the test with one user, PyTorch had a very small performance advantage over Triton (PyTorch) with 34.16 RPS and 33.14 RPS, respectively. From 10 users and higher, Table 10 in the Appendix shows us that all three frameworks' throughputs stay constant, implying that this is caused by the models themselves, and not by the user count.

It should be mentioned that although we are comparing the frameworks to each other, we should also consider the absolute values of the throughputs. Triton (TensorFlow) being able to concurrently process 24 RPS consistently no matter how many users are firing requests at the framework is no small feat. This is much more significant when considering the performance of Triton (PyTorch), which processes almost double the requests. Overall,

we can say that the throughput of the frameworks when observing them individually is excellent.

User Count Framework	1	10	25	50	75	100
PyTorch	**28.51**	280.14	685.04	1370.44	2045.64	2696.93
Triton (PyTorch)	29.62	**224.50**	**556.97**	**1112.66**	**1676.74**	**2190.66**
Triton (TensorFlow)	92.43	465.68	1159.55	2287.42	3415.39	4541.18

Table 3: Prediction latency of the Wav2Vec2 model (in milliseconds)

An interesting observation is that Triton's (TensorFlow) prediction latency is (in comparison) quite high, starting from the test with only one virtual user. This gets considerably worse the more users are added to the load test, peaking at almost double that of Triton's (PyTorch) prediction latency in the test with 100 concurrent users.

This being said, in the test with one user, all frameworks show a very small response time for most requests (under 100 ms). This makes it quite applicable to the task of real-time text-to-speech transcription. However, as the concurrent users requesting the ASR service increase, we can see the latency dramatically increasing. Although the processing time for each individual request remains relatively similar, the latency is higher due to their waiting time in the queue. This shows that under heavy user load, performance may degrade to the extent that the service becomes unusable. Although this should be taken with a grain of salt, since in a real-world scenario, the platform is not going to constantly be under the heaviest load.

5.1.3 Wav2Vec2 truncated

The truncated version of Wav2Vec2 was implemented to accommodate the limitations of the official TensorFlow implementation of the Wav2Vec2 model. This workaround involved truncating (or padding) the inputs for all frameworks to be able to continue our performance evaluation across all four frameworks. This approach results in incomplete outputs, as the truncated inputs do not provide the full context necessary for full model predictions, resulting in sentences seemingly cutting off, sometimes mid-word. Despite these obstacles, we believe the analysis offers interesting results into the performances of the different frameworks.

Figure 3 gives an overview of the throughput results for the Wav2Vec2 truncated model. This model exhibits the same behavior as was observed in the original Wav2Vec2 model. Triton (PyTorch) consistently outperforms all other frameworks in both throughput and latency. The truncated inputs, which are significantly shorter than the normal Wav2Vec2 model inputs, reinforce the evidence from the normal Wav2Vec2 model that NVIDIA Triton provides much better performance. Here we can see the same concept as was seen in the original Wav2Vec2 model above, namely the throughput plateauing in the tests with more than 10 users. This time, however, due to the input being significantly shorter, this value is higher. Interestingly though, we can observe that TensorFlow performs slightly better than PyTorch and much better than Triton (TensorFlow), which is quite a difference from the behavior observed in the ResNet50 test.

When considering the throughput values for each framework individually, we can see

that the values are 1.5 to 2 times better than their respective values from the original Wav2Vec2 model. However, we believe it is impossible to reach a consensus of whether these values would be useful for any real-world application. We make this claim due to the fact that all the inputs were truncated in order to fit this model, meaning a large number of them were incomplete.

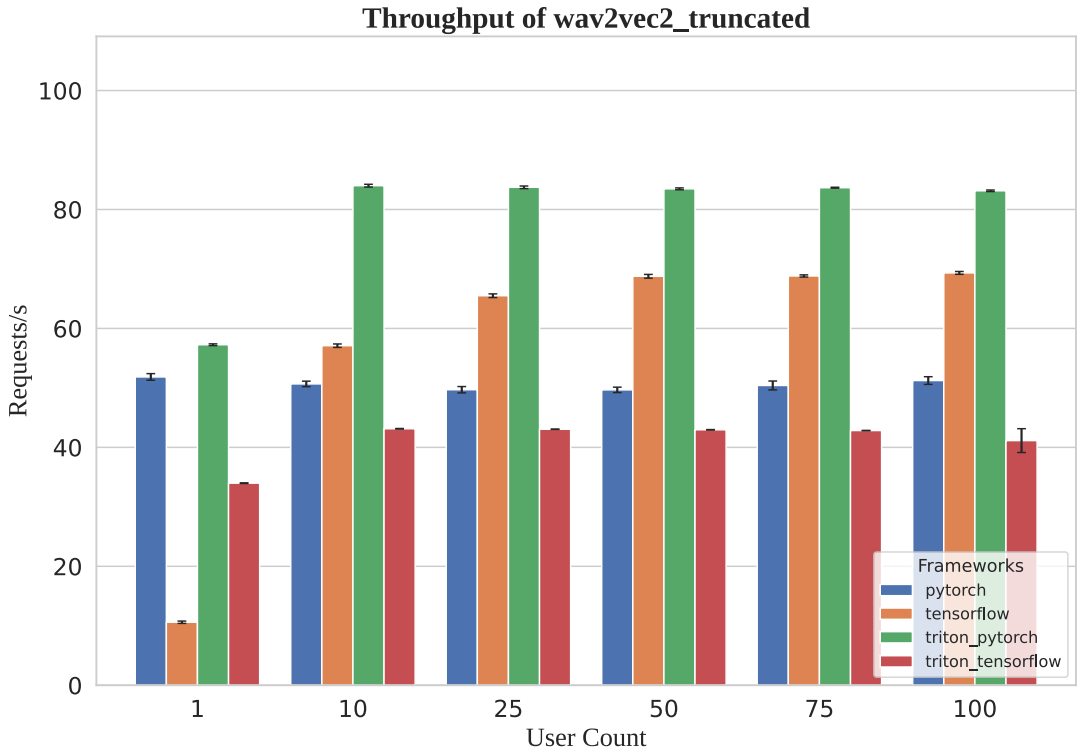

Figure 3: Throughput of the Wav2Vec2 Truncated model (in requests per second)

Moving on to latency, the graphs in Table 4 shows us that when considering only one user, TensorFlow has the worst latency, although this improves considerably, jumping to second-best when the system is under load at 100 virtual users. Triton (PyTorch) once again has the shortest request time throughout all user tests.

User Count Framework	1	10	25	50	75	100
PyTorch	18.33	195.26	484.33	965.58	1430.13	1885.84
TensorFlow	54.20	130.30	305.57	673.08	1019.39	1358.25
Triton (PyTorch)	**16.92**	**110.44**	**276.10**	**551.70**	**823.64**	**1100.21**
Triton (TensorFlow)	28.87	215.43	536.20	1068.26	1596.83	2133.12

Table 4: Prediction latency of the Wav2Vec2 (Truncated) model (in milliseconds)

The latency values here show nearly a two-fold improvement from the original Wav2Vec2 model, although this is to be expected due to the inputs' size. However, since the inputs are truncated and incomplete, we believe that there are no realistic applications suited for them. The obtained results are very good for comparing the frameworks to each other, but we cannot tie the results to any of our defined use cases.

5.1.4 BART Large CNN

For BART Large CNN, only PyTorch was able to be evaluated. The results for both throughput and latency were significantly worse compared to other models, which is expected given the model's larger size. Due to TorchServe's working queue that can handle up to 100 requests, the experiments with BART showed no failure rate. Unsurprisingly, additional tests show that any user count above 100 means that all other requests get immediately rejected by TorchServe. Because it cannot keep more than 100 requests in the queue, and the inference takes a long time, we see that every single request receives a response with code 503, telling us that there are no more available workers for the model.[21]

The performance of PyTorch here was constant. Throughout all the tests, the framework maintained a throughput of around 1.42 RPS, with variations only within the range of ±0.02 RPS. We can, however, discern that the latency immensely increases the more we raise the number of users submitting requests. As we can see in Table 5, the average latency for the test with only one virtual user is 695 ms.

User Count Framework	1	10	25	50	75	100
PyTorch	695.00	6795.64	14962.30	24651.70	29340.88	29842.83

Table 5: Prediction latency of the BART model (in milliseconds)

The throughput remaining constant is a very good sign for TorchServe, showing that this is due to the performance of the model, and not only the framework's performance. When considering real-world use cases, such as **UC2**, we must also look at the latency. At the heaviest load, each individual request takes averagely half a minute to receive a response. This is simply unviable for any realistic system, since the long waiting time would be unacceptable for any user. We should once again mention that realistically, most systems will not constantly be under the heaviest load.

5.2 Usability Results

This study used specific measures to assess the usability of the three chosen DL frameworks. Each system got a score from 1 to 5 in these areas. The ratings were based on our experience with the framework, and observations to give a detailed look at what each system does well, as well as where they could improve when used in real situations.

Getting some models running was quite tricky, due to the fact that all of the frameworks have different methods of serving their models. As we saw in the previous section, not all of the chosen models were able to be run in all frameworks, although all of them had official implementations in their respective frameworks.

[21] *Troubleshooting guide - PyTorch/Serve master documentation.* URL: https://pytorch.org/serve/Troubleshooting.html#inference-request-failed-with-exception-serviceunavailableexception-error-code-503 (visited on 21/06/2024)

5.2.1 User-Friendliness

TorchServe Each of the frameworks has a different method of serving. TorchServe expects binary data, meaning the user can run inference on the framework via an HTTP or gRPC request. Its management API allows users to easily manage models, including registering, updating, and unregistering models on the fly.

Triton Inference Server Triton Inference server requires its own client to be set up to run inference on the models. It also needs a particular model repository[22] layout, which requires precise file paths of the model files, as well as a configuration file. This is necessary due to the fact that Triton can serve models from multiple frameworks, although it does complicate the serving process.

TensorFlow Serving TensorFlow Serving expects the user to download the Serving docker container[23], and serve the models with it. As a consequence, most of the documentation focuses on this default serving method, which makes the process of running TensorFlow Serving natively quite difficult. It expects its inputs for the model to be in a serializable JavaScript Object Notation (JSON) format, such as NumPy arrays. As a result, additional preprocessing time is needed for data deserialization once the input arrives.

5.2.2 Documentation Quality

TorchServe The documentation quality of TorchServe is quite good. It includes pages ranging from a quick setup guide to more complex topics, along with a Frequently Asked Questions section. The documentation is updated weekly, with recent updates reflecting the latest features and compatibility improvements, along with detailed configuration options.

Triton Inference Server Triton Inference Server, in comparison, delves even deeper than TorchServe. Its documentation provides an extensive insight into the inner workings of the Triton architecture and the functionality of each component. The documentation is also up-to-date, with new entries bi-weekly. providing comprehensive guides and tutorials that reflect recent updates and new features in the Triton ecosystem.

TensorFlow Serving TensorFlow Serving caused a few issues that stem from the unclarity of the documentation. Firstly, the documentation is split into multiple sources. The main documentation is on the TensorFlow website,[24] however, there are also tutorials and guides on the Keras website.[25] We think this is because some official model implementations come from Keras, and others from Kaggle Models (formerly TF Hub)[17]. The content of its documentation is also not very extensive, although it contains articles on

[22] *Model Repository - NVIDIA Triton Inference Server.* URL: https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/model_repository.html (visited on 20/06/2024)

[23] *tensorflow/serving - Docker Image | Docker Hub.* URL: https://hub.docker.com/r/tensorflow/serving

[24] *Serving Models | TFX | TensorFlow.* URL: https://www.tensorflow.org/tfx/guide/serving (visited on 20/06/2024)

[25] *Serving TensorFlow models with TF Serving* URL: https://keras.io/examples/keras_recipes/tf_serving/ (visited on 20/06/2024)

specific use cases. It is also unclear how often the TensorFlow Serving documentation is updated.

5.2.3 Project Features

TorchServe TorchServe provides two very useful features, namely its reliance on binary inference data, and custom model handlers. They provide a higher degree of customization which is absent from the other two frameworks. This combination allows users to connect with the APIs to interact with the model server and keep the work required to a minimum. It reduced the effort required to set up and debug issues.

Triton Inference Server Triton Inference Server stands out in this regard, since it is able to serve models of different frameworks. Other than PyTorch and TensorFlow, Triton also serves models that have been converted to Open Neural Network Exchange (ONNX) format, TensorRT models, Caffe2, and others. Furthermore, Triton implements model ensembling, a feature it shares with TorchServe. Model ensembling involves chaining together multiple models or pre-/postprocessing steps into a single pipeline for creating predictions.

TensorFlow Serving TensorFlow Serving offers features like versioning for simultaneous model testing, deployment through containerization via Docker and Kubernetes, and optimization options such as batching and caching for efficient inference handling. It also includes built-in monitoring capabilities to track model performance and server health metrics.

5.2.4 Community Support

TorchServe TorchServe is built and maintained by Amazon Web Services (AWS) in collaboration with Facebook/Meta[26]. In the PyTorch Github repository, issues are generally responded to in a week, with some issues getting attention within a few days. As of 20/06/2024, 21 out of 59 total issues are resolved.

Triton Inference Server Triton Inference Server, being an official project from NVIDIA, likely benefits from dedicated full-time employees working on its development and maintenance. Although it has a smaller community compared to TensorFlow and PyTorch, there are still discussions available on forums and GitHub. There are a lot more issues on the GitHub page, with 56 out of 187 of them being resolved.

TensorFlow Serving TensorFlow Serving has the largest number of contributors (214) and a large amount of resources available for support. It is a framework created by Google, however, the extent to which Google maintains the project remains unclear. The issue response time varies, but recent issues indicate that the maintainers are quite responsive, with many issues addressed within days. As of 20/06/2024, only 10 issues have been opened, with two of them being resolved.

[26] *Introducing TorchServe: a PyTorch model serving framework* URL: https://aws.amazon.com/about-aws/whats-new/2020/04/introducing-torchserve/ (visited on 24/06/2024)

5.2.5 Maintenance and Update Frequency

TorchServe PyTorch TorchServe is currently still on major release 0. It follows a quarterly release cycle with about 4 releases in the last year, aligning major updates with PyTorch releases.

Triton Inference Server NVIDIA Triton Inference Server has the most constant releases, with around 12 of them in the past year, maintaining a regular schedule towards the end of the month.

TensorFlow Serving TensorFlow Serving follows a less predictable release cycle, due to the fact that it is aligned with the releases of TensorFlow itself. There have been approximately 15 releases in the past year, including minor updates and consistent activity. Minor and patch updates are released as needed to address specific issues or dependencies, often first solved in the nightly builds. Overall, TensorFlow Serving and Triton Inference Server exhibit more frequent updates compared to PyTorch Serve.

5.3 Evaluation

5.3.1 Performance

In Table 6, we clearly show the latency performance of all the frameworks, and highlight the best performing frameworks. We can see that PyTorch is overall the best when measuring prediction latency. Also referring to Figure 1 and 5, we can show that the range of the latency results for all (but one) frameworks was quite stable, thus proving that the results are accurate.

Frameworks	PyTorch Torchserve	TensorFlow Serving	Triton (PyTorch)	Triton (TensorFlow)
ResNet50	**11.29**	21.43	12.87	16.89
Wav2Vec2	**28.51**	-	29.62	92.43
Wav2Vec2 (truncated)	18.33	54.20	**16.92**	28.87
BART	**695.00**	-	-	-

Table 6: Frameworks' performance, based on latency at 1 user (in milliseconds)

Regarding the throughput, the results were consistent across different user counts, with PyTorch and Triton (PyTorch) competing closely for first place. The throughput performance table in Table 7 also points to the fact that the performance of TensorFlow and Triton (TensorFlow) is consistently the worst, in most cases performing around 2x worse than the best framework. Nevertheless, Triton Inference Server showed a good capability to increase the prediction speed for TensorFlow models, bringing it closer to that of PyTorch.

Considering these performance evaluations, the clear contenders for addressing **RQ1** would be either PyTorch or Triton (PyTorch). PyTorch consistently shows the lowest latency in the chosen models, making it ideal for real-time applications, with Triton (PyTorch) following closely. Whereas PyTorch excels in low latency, Triton (PyTorch) stands out for high throughput across all models, only trailing behind the former framework.

Frameworks	PyTorch Torchserve	TensorFlow Serving	Triton (PyTorch)	Triton (TensorFlow)
ResNet50	**104.10**	40.59	99.74	74.53
Wav2Vec2	35.16	-	**41.43**	24.51
Wav2Vec2 (truncated)	51.24	69.34	**83.14**	41.14
BART	**1.4412**	-	-	-

Table 7: Frameworks' performance, based on throughput at 100 users (in RPS)

5.3.2 Usability

Usability scores ranging from 1-5 were assigned to each serving framework based on our personal experience and observations. The scores align with the usability criteria detailed in Section 4.3. A higher score indicates better usability, with a score of 5 given to a framework that is intuitive to set up and deploy, with comprehensive and understandable documentation, active community support, helpful features for model customization, and frequent updates. Conversely, a score of 1 would indicate significant problems in usability, such as convoluted setup processes, incompatibility with specific models, outdated documentation, no community engagement, or infrequent updates.

Frameworks	PyTorch Torchserve	TensorFlow Serving	NVIDIA Triton
User-Friendliness	5	3	4
Documentation Quality	5	3	5
Project Features	5	4	5
Community support	4	5	4
Maintenance and Update Frequency	4	5	5

Table 8: Usability scores (1-5). Higher scores indicate better usability.

The scores shown in Table 8 are crucial for adressing **RQ2**. PyTorch emerges as the most usable framework, scoring nearly perfect across all criteria. Its relatively easy setup, comprehensive documentation, and features set it apart from the other frameworks.

Section Summary This section has presented the findings from our performance tests and our usability analysis. PyTorch consistently outperforms TensorFlow and Triton, although TensorFlow shows significant improvement over past benchmarks. In terms of usability, PyTorch's key takeaway is its great user-friendliness, TensorFlow's is its larger community support, whereas Triton has the most unique project features. This two-step analysis provides a holistic view of each framework, further helping our understanding of their respective strengths and weaknesses.

6 Discussion

This section discusses the main findings of this research. Firstly, Section 6.1 addresses the challenges faced during the set up and execution of the experiments. In Section 6.2, it delves into the interpretation of the results obtained from the experiments by tying them back to our earlier defined use cases, showing how the different inference frameworks impact practical scenarios and real-world applications.

6.1 Challenges

6.1.1 Challenges with Wav2Vec2 in TensorFlow

As mentioned above, TensorFlow could not serve Wav2Vec2 due to its implementation in TensorFlow Hub[27], it violated our usability constraints because the SavedModel's serving signature is constrained, and does not allow inputs of a larger size. Specifically, the model requires an input tensor with dimensions `(-1, 50000)`, meaning each audio input sequence must have 50,000 samples. Initially, we tried defining our own input format, but it did not work. We are unsure why, but our investigation suggests it might be due to a specific operation in the TensorFlow graph of the Wav2Vec2 model. This issue seems to occur during the convolutional layer in the positional convolutional embedding `(pos_conv_embed)` of the Wav2Vec2 encoder.

After numerous unsuccessful attempts, we decided to stop pursuing this approach. Our solution is to ensure all frameworks adhere to the constraint of the TensorFlow model, which we defined in Section 5.1.3. Since we are only feeding the model inputs of batch size 1, all we have to do to fit the required size of `(-1, 50000)` is truncate or pad the processed audio). Although this workaround allows for performance evaluation across frameworks, it introduces certain inaccuracies. This (unusual) truncation process itself should ideally not be included in the performance metrics, as it is part of preprocessing rather than model inference. However, since we are looking for an overall comparison of the entire model inference pipeline, and since as shown in Listing 2, all of the inputs are being truncated in the same method, we are considering this step part of the preprocessing.

6.1.2 Challenges with serving BART

Triton typically requires models to be converted to TorchScript via tracing to be able to process a model's inputs and outputs in a static, predictable way. To achieve this, PyTorch leverages its JIT compiler to translate a subset of Python programs into a representation that can have optimizations run on it and can get interpreted by the TorchScript interpreter at runtime.

However, tracing dynamic operations, such as the `.generate()` function of the BART model, may be difficult or even impossible due to the nature of the JIT compiler in PyTorch. The problem here lies with the `.generate()` method itself, which has loops of variable length and control flows that are not handled well by `torch.jit.trace`. This method of tracing essentially captures the operations executed over a single forward pass

[27] *Kaggle | Wav2Vec2.* `URL:` https://www.kaggle.com/models/kaggle/wav2vec2/tensorFlow2/960h/1?tfhub-redirect=true (visited on 18/06/2024)

to create a static graph, hence failing to correctly trace operations with dynamic control flow.

> *Tracing only correctly records functions and modules that are not data-dependent (e.g., do not have conditionals on data in tensors) and do not have any untracked external dependencies (e.g., perform input/output or access global variables). Tracing only records operations done when the given function is run on the given tensors. Therefore, the returned ScriptModule will always run the same traced graph on any input. This has some important implications when your module is expected to run different sets of operations, depending on the input and/or the module state.* [Dev24]

Serving the BART model via TensorFlow gave us challenges similar to those encountered with the Wav2Vec2 model. The main difficulty arose when trying to serve the BART model in SavedModel format with a custom signature that utilized the `.generate()` method from HuggingFace's implementation.

TensorFlow Serving requires models to be saved in the SavedModel format, which includes defined signatures for serving. However, as we mentioned above, the `.generate()` method in BART involves loops and conditionals that are not easily captured in a static graph format. After exporting the model with a custom serving signature, TensorFlow Serving encountered issues with serving it. Based on the TensorFlow Serving logs, and the issue we created on GitHub[28], we believe the problem stems from the creation of dynamic tensors within the method itself. This poses challenges for Accelerated Linear Algebra (XLA)[29], the compiler that optimizes the runtime of TensorFlow models.

The inability to serve BART with TensorFlow shows an important limitation: The static graph requirements imposed by these frameworks limit their ability to handle models with complex control flows, such as generative models. In the end, we were only able to serve the BART model with PyTorch, as TorchServe does not require the model to be explicitly compiled with JIT. It also provides the ability to write custom model handlers, allowing us to call the `.generate()` method during the handling step, after preprocessing.

6.2 Interpretation of the results

Our results suggest that, although there have been quite a few differences from the Biano-AI research [AI21], the best-performing framework is still PyTorch. Nevertheless, we can see that none of the tests showed any failed requests, different from the preceding research. The individual performance of all frameworks has also improved considerably, with TensorFlow Serving improving the most. This suggests that ongoing updates and community contributions are improving its capabilities, even though it still lags behind PyTorch in some aspects.

Based on the results in Tables 6 and 7, it is evident that the performance of the ML inference frameworks varies significantly, with each framework exhibiting strengths and weaknesses in different areas.

PyTorch consistently demonstrates lower latency compared to other frameworks. This

[28] *Issue #2217 · tensorflow/serving · GitHub* URL: https://github.com/tensorflow/serving/issues/2217 (visited on 19/06/2024)

[29] *OpenXLA Project.* URL: https://openxla.org/xla (visited on 19/06/2024)

lower latency makes PyTorch a highly suitable choice for applications where real-time predictions are crucial, such as in ASR tasks. Additionally, PyTorch exhibits high throughput, meaning it can handle a large number of requests per second without significant performance degradation. This makes PyTorch ideal for high-demand applications where both low latency and high throughput are required.

TensorFlow, on the other hand, generally shows higher latency compared to PyTorch and Triton (PyTorch). This higher latency might be a limiting factor for applications that require immediate responses. However, TensorFlow's throughput is competitive in the ASR task, especially in scenarios with high user counts. This indicates that TensorFlow can still be effective in applications where throughput is prioritized over latency, making it a viable option for certain types of high-demand environments.

Triton Inference Server, when using PyTorch models, shows competitive latency similar to native PyTorch. This makes it another strong contender for real-time applications. Triton's throughput, particularly with PyTorch models, is the highest among the frameworks evaluated. This high throughput makes Triton highly suitable for applications that need to handle very high demand. Even when using TensorFlow models, Triton shows improved latency and throughput compared to native TensorFlow, making it a better choice for TensorFlow-based applications that require higher performance.

To apply the results, we must also understand the metrics. Our use cases pertain more to the relationship with the results, rather than the results proper. This is why we discuss them in this section, rather than Section 5. Let us consider **UC2**. For this use case, the important aspect would be to offer the users that want AIaaS a response from the model as soon as possible. When considering a real scenario like this, the serving framework might not be under load constantly, which means one of the main goals would be to offer as low of a latency as possible. This is why when we look at the results, we consider the latency concerning the test with a single virtual user. This is also why we would recommend the choice of either PyTorch or Triton (PyTorch), instead of the (much) slower TensorFlow Serving. However, for applications where TensorFlow's ecosystem and tools are needed, its performance may still be acceptable, especially given the significant community support and documentation available.

In **UC1**, where usability is paramount for users with limited technical expertise, the choice of serving framework extends beyond performance metrics alone. Ease of setup, deployment, and maintenance are critical factors influencing usability. We saw PyTorch being the best option here, scoring the highest in our usability scores (see Table 8). Its comprehensive documentation and community support ensure that users can deploy models with minimal issues and easily troubleshoot problems that arise. It should be mentioned that Triton also performs well, particularly in serving models from other frameworks and offering many features for optimization.

Section Summary In this section, we have brought attention to the major problems plaguing TensorFlow Serving and Triton Inference Server, namely their problems with Serving Signatures and static graph requirements, respectively. We have also interpreted the results, discussing their implications for the selection of inference frameworks. PyTorch's superior performance and user-friendliness make it the preferred choice, though Triton's flexibility and TensorFlow's extensive ecosystem are also valuable. This section emphasizes the importance of aligning framework choice with specific requirements.

7 Conclusion

This thesis has provided a quantitative and qualitative comparison of various ML inference frameworks. The research question aimed to identify the most suitable framework for different use cases based on performance and usability metrics.

The methodology involved a carefully constructed approach to ensure the reliability of our findings. We selected representative models for three distinct tasks. Each model was deployed and tested on the respective frameworks under controlled conditions. Our experiments included two different types of load-tests: single-stream and multi-stream. Both performance and usability were assessed based on clear, concise criteria that we constructed.

Our performance evaluation revealed that PyTorch consistently outperformed the other frameworks in terms of latency and throughput across different models, although Triton followed closely behind. Specifically, PyTorch demonstrated the lowest latency for the image classification and ASR tasks, making it an excellent choice for real-time applications.

Moving on from the performance evaluation, the usability scores show that PyTorch came out on top, although once again Triton came in close second. Its setup process, broad documentation, and active community support contributed to this high score. When considering TensorFlow's usability, however, we can see that it was hindered by vague setup procedures and unclear documentation.

This study's contribution lies in its in-depth analysis of the ML serving frameworks, providing valuable insights for different use cases and applications. By evaluating both performance and usability, this research shows that the choice of serving framework is as critical as the selection of the model for ML tasks, proving that the serving framework can significantly impact the overall effectiveness and efficiency of the deployed model.

Our study was limited to the default configurations of the frameworks and models, therefore future work should include testing the ML models without any constraints, and exploring which frameworks would be the most effective at running the different models. Other than that, expanding the scope of future work to investigate performance across more diverse use cases, or to include novel frameworks, such as in edge computing, could provide valuable insights into the field of ML.

In conclusion, this thesis has provided a thorough comparison of TensorFlow Serving, PyTorch TorchServe, and NVIDIA Triton Inference Server, showcasing their strengths and weaknesses in different scenarios. The insights gained from this research can guide users to select the most suitable framework based on particular requirements.

References

[AI21] Biano AI. *Quantitative Comparison of Serving Platforms for Neural Networks*. Website. Accessed: 2024-06-20. Aug. 2021. URL: https://biano-ai.github.io/research/2021/08/16/quantitative-comparison-of-serving-platforms-for-neural-networks.html.

[Bae+20] Alexei Baevski et al. "wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations". In: *CoRR* abs/2006.11477 (2020). arXiv: 2006.11477. URL: https://arxiv.org/abs/2006.11477.

[BH22] Hojat Behrooz and Yeganeh M. Hayeri. "Machine Learning Applications in Surface Transportation Systems: A Literature Review". In: *Applied Sciences* 12.18 (2022). ISSN: 2076-3417. DOI: 10.3390/app12189156. URL: https://www.mdpi.com/2076-3417/12/18/9156.

[Cha+19] Ryan Chard et al. "DLHub: Model and Data Serving for Science". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 283–292. DOI: 10.1109/IPDPS.2019.00038.

[Cho18] Aditya Chopra. "Exploring Novel Architectures For Serving Machine Learning Models". In: 2018. URL: https://api.semanticscholar.org/CorpusID:51932875.

[CR19] Jiasi Chen and Xukan Ran. "Deep Learning With Edge Computing: A Review". In: *Proceedings of the IEEE* 107.8 (2019), pp. 1655–1674. DOI: 10.1109/JPROC.2019.2921977.

[Cra+17] Daniel Crankshaw et al. "Clipper: A {Low-Latency} online prediction serving system". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, p. 615.

[Dev+18] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[Dev24] PyTorch Developers. *torch.jit.trace | PyTorch 2.3 documentation*. Accessed: 19-June-2024. 2024. URL: https://pytorch.org/docs/stable/generated/torch.jit.trace.html.

[DGN17] Utsav Drolia, Katherine Guo, and Priya Narasimhan. "Precog: prefetching for image recognition applications at the edge". In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. SEC '17. San Jose, California: Association for Computing Machinery, 2017. ISBN: 9781450350877. DOI: 10.1145/3132211.3134456.

[DHB20] Matthew F. Dixon, Igor Halperin, and Paul Bilokon. *Machine Learning in Finance: From Theory to Practice*. Springer International Publishing, 2020. Chap. 1. Introduction, pp. 3–40. ISBN: 9783030410681. DOI: 10.1007/978-3-030-41068-1. URL: http://dx.doi.org/10.1007/978-3-030-41068-1.

[He+15] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[Her+15] Karl Moritz Hermann et al. "Teaching Machines to Read and Comprehend". In: *CoRR* abs/1506.03340 (2015). arXiv: 1506.03340. URL: http://arxiv.org/abs/1506.03340.

[Jav+22] Mohd Javaid et al. "Significance of machine learning in healthcare: Features, pillars and applications". In: *International Journal of Intelligent Networks* 3 (2022), pp. 58–73. ISSN: 2666-6030. DOI: https://doi.org/10.1016/j.ijin.2022.05.002. URL: https://www.sciencedirect.com/science/article/pii/S2666603022000069.

[Jia+20] Xiaoqi Jiao et al. "TinyBERT: Distilling BERT for Natural Language Understanding". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Ed. by Trevor Cohn, Yulan He, and Yang Liu. Online: Association for Computational Linguistics, Nov. 2020, pp. 4163–4174. DOI: 10.18653/v1/2020.findings-emnlp.372. URL: https://aclanthology.org/2020.findings-emnlp.372.

[Lew+19] Mike Lewis et al. "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension". In: *CoRR* abs/1910.13461 (2019). arXiv: 1910.13461. URL: http://arxiv.org/abs/1910.13461.

[Ols+17] Christopher Olston et al. "TensorFlow-Serving: Flexible, High-Performance ML Serving". In: *CoRR* abs/1712.06139 (2017). arXiv: 1712.06139. URL: http://arxiv.org/abs/1712.06139.

[Pan+15] Vassil Panayotov et al. "Librispeech: An ASR corpus based on public domain audio books". In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2015, pp. 5206–5210. DOI: 10.1109/ICASSP.2015.7178964.

[Ran+18] Xukan Ran et al. "DeepDecision: A Mobile Deep Learning Framework for Edge Video Analytics". In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 2018, pp. 1421–1429. DOI: 10.1109/INFOCOM.2018.8485905.

[Red+19] Vijay Janapa Reddi et al. "MLPerf Inference Benchmark". In: *Computing Research Repository (CoRR)* abs/1911.02549 (2019). arXiv: 1911.02549. URL: http://arxiv.org/abs/1911.02549.

[Red+21] Vijay Janapa Reddi et al. "The Vision Behind MLPerf: Understanding AI Inference Performance". In: *IEEE Micro* 41.3 (2021), pp. 10–18. DOI: 10.1109/MM.2021.3066343.

[Shi+16] Weisong Shi et al. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–638. DOI: 10.1109/JIOT.2016.2579198.

[Vas+17] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

A Appendix

A.1 Listings

```
desired_length = 50000
current_length = triton_inputs.shape[1]
padding_length = max(0, desired_length - current_length)
truncation_length = max(0, current_length - desired_length)
if padding_length > 0:
    triton_inputs = np.pad(triton_inputs, ((0, 0), (0, padding_length)))
elif truncation_length > 0:
    triton_inputs = triton_inputs[:, :desired_length]
```

Listing 2: Truncating of the inputs for the wav2vec2 model (Triton)

A.2 Figures

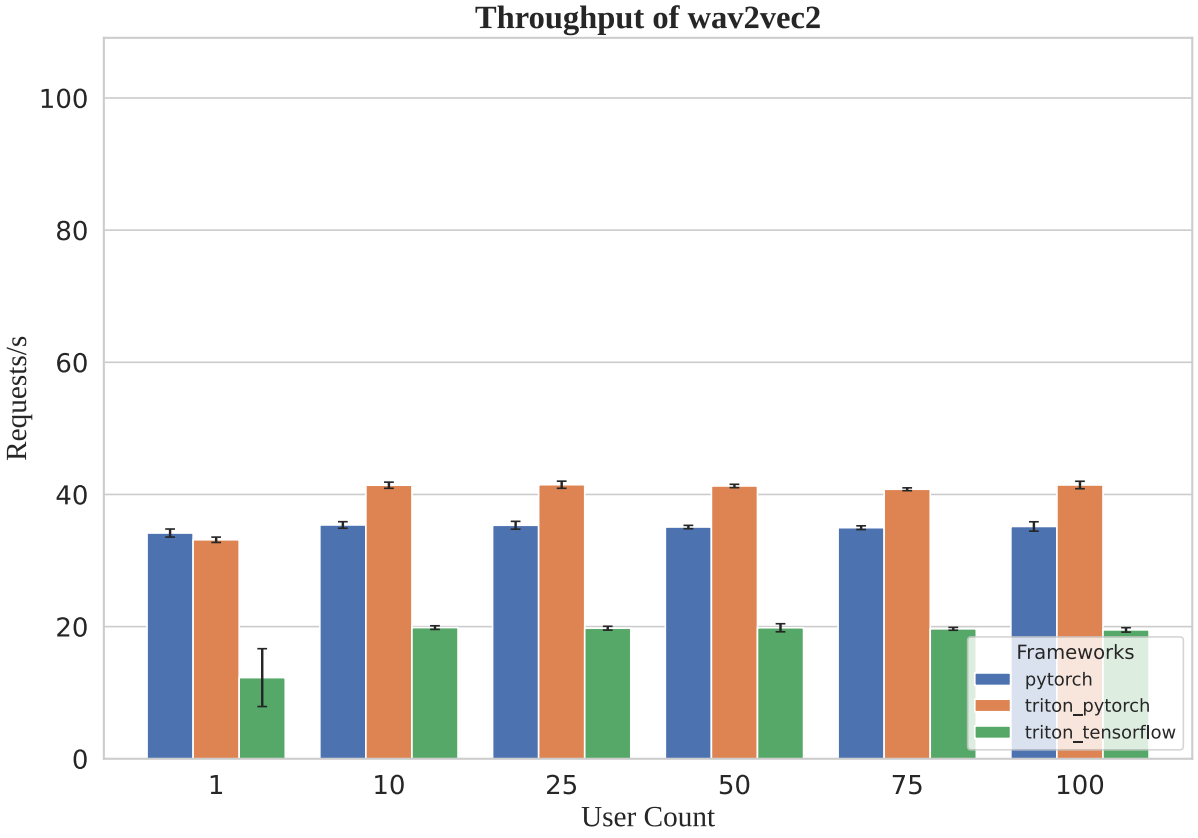

Figure 4: Throughput of the Wav2vec2 model

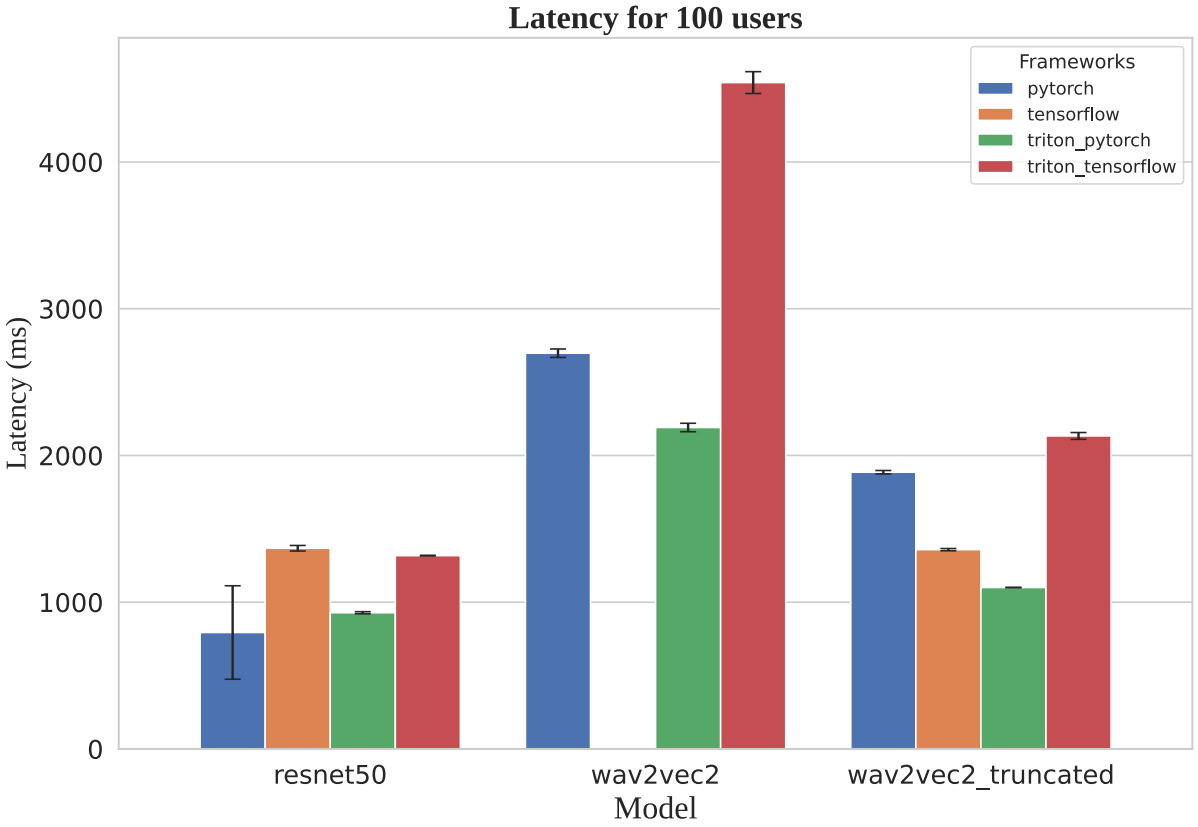

Figure 5: Latency results for 100 virtual users

A.3 Tables

User Count Framework	1	10	25	50	75	100
PyTorch	**11.29**	95.37	242.59	477.26	709.51	**793.68**
TensorFlow	21.43	**32.26**	**77.37**	575.80	983.26	1367.72
Triton (PyTorch)	12.87	82.76	197.41	**431.56**	**681.01**	928.29
Triton (TensorFlow)	16.89	130.93	328.86	660.60	985.50	1317.58

Table 9: Prediction latency of the ResNet50 model

User Count Framework	1	10	25	50	75	100
PyTorch	**34.16**	35.38	35.34	35.07	34.97	35.16
Triton (PyTorch)	33.14	**41.40**	**41.47**	**41.28**	**40.79**	**41.43**
Triton (TensorFlow)	6.55	21.85	24.62	24.34	24.45	24.51

Table 10: Throughput of the Wav2Vec2 model

User Count Framework	1	10	25	50	75	100
PyTorch	51.84	50.67	49.69	49.67	50.41	51.24
TensorFlow	10.60	57.10	65.50	68.77	68.82	69.34
Triton (PyTorch)	**57.26**	**84.00**	**83.73**	**83.47**	**83.66**	**83.14**
Triton (TensorFlow)	33.98	43.13	43.05	42.96	42.83	41.14

Table 11: Throughput of the Wav2Vec2 (Truncated) model

User Count Framework	1	10	25	50	75	100
PyTorch	1.3942	1.4142	1.4139	1.4433	1.4465	1.4412

Table 12: Throughput of the BART model (in requests per second)