A Quantitative and Qualitative Comparison of Machine Learning Inference Frameworks

Egi Brako
Department of Computer Science
Georg-August-Universität Göttingen
Göttingen, Germany
e-mail: egi.brako@gmail.com

Jonathan Decker
Department of Computer Science
Georg-August-Universität Göttingen
Göttingen, Germany

Julian Kunkel
Department of Computer Science
Georg-August-Universität Göttingen
Göttingen, Germany

***Abstract*—As Artificial Intelligence (AI) continues to advance and impact diverse fields, ensuring universal access to its abilities becomes increasingly crucial. To make AI models accessible to users, 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 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, analyse 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 find that some platforms have 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.**

Keywords-artificial intelligence; inference engines; machine learning;

I. Introduction

Machine Learning (ML) has emerged as a pivotal technology across various domains, revolutionizing industries such as transportation, healthcare, and finance. With the growing reliance on ML models for critical decision-making and everyday applications, the need to effectively serve these models to a wider audience has become increasingly important. Model serving refers to the deployment, management, and maintenance of ML models in production environments, ensuring their availability, responsiveness, and accuracy in delivering predictions or results in real-time.

Serving frameworks, also known as inference frameworks, play a crucial role in this process by facilitating the deployment of ML models at scale. These frameworks manage multiple requests, optimize computational resources, and enable seamless model updates. Despite their significance, selecting the right serving framework remains a challenge due to varying requirements and the distinct features each framework offers.

This paper aims to provide a comprehensive comparison of three popular ML serving frameworks: TorchServe[1], TensorFlow Serving[2], and Triton Inference Server[3]. By evaluating both performance metrics, such as latency and throughput, and usability factors, including user-friendliness and documentation quality, this research seeks to identify the most suitable framework for different ML tasks.

The remainder of the paper is organized as follows: Section II reviews related works, highlighting previous studies, advancements, as well as gaps in the field. Section III details the methodology, including the experimental setup and evaluation metrics. The results of the experiments are then presented in Section IV followed by an evaluation and discussion of the findings in Section V. Finally, Section VI concludes the paper with a summary of the research and suggestions for future work.

II. 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. Previous studies, such as *MLPerf Inference Benchmark* [4] and its succeeding research *The Vision Behind MLPerf: Understanding AI Inference Performance* [5], have made significant strides in establishing benchmarks for ML inference, focusing on key performance metrics and trade-offs between accuracy and performance. These works laid the groundwork for evaluating ML models across various applications, though they were limited by the scope of models and scenarios considered. For instance, models like Bidirectional Encoder Representations from Transformers (BERT) [6] and transformers [7] were not included, leaving room for further exploration of these applications.

In the area of custom serving systems, works like *Clipper: A {Low-Latency} online prediction serving system* [8] provided foundational insights into real-time ML predictions, with a focus on modularity and performance optimization techniques. However, Clipper's evaluation was limited to specific benchmarks, and it is no longer maintained, making its findings somewhat outdated.

Edge computing research, as reviewed in *Deep Learning With Edge Computing: A Review* [9] or *Edge Computing: Vision and Challenges* [10] has also been extensive, focusing on bringing Deep Learning computations closer to end devices for tasks like computer vision and Natural Language Processing (NLP). While valuable, this body of work is more concerned with

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2024. ISBN: 978-1-68558-216-6

inference on edge devices rather than general-purpose serving frameworks.

Our research fills the gap by providing a comprehensive, up-to-date comparison of widely-used ML serving frameworks, offering critical insights for selecting the most suitable platform for diverse ML applications.

III. Methodology

This research aims to evaluate the performance and usability of different machine learning inference 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". This has led us to examine both aspects separately, and to form individual evaluations for them. Both performance metrics and the practical, user-oriented aspects contribute to determining how useful these frameworks are, under various conditions, for different users.

A. Performance

For the performance evaluation, we 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 will keep the platform under constant heavy load, simulating a worst-case scenario helps ensure the system is reliable, even under heavily demanding circumstances. On the other hand, 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. In both of these scenarios, three different metrics will be measured, namely: Throughput (in Requests per second), Latency (in milliseconds), and Failures (in failures per second). These measurements will help us understand how well the systems work in different situations.

B. Usability

For the usability evaluation, we must define formal metrics and strictly evaluate the serving frameworks, as it is difficult to conduct without set criteria. 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 criteria, which will serve as our reference parameters throughout our analysis.

The reference parameters throughout our analysis will be five loose criteria that we have defined, which we will use to judge the frameworks' qualitative characteristics. Namely, these criteria are User-Friendliness, Documentation Quality, Project Features, Community Support, and Maintainance and Update Frequency. They will all receive a score based on our experience with the framework ranging from 1 to 5. 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. Through this, a systematic comparison can be made based on the observations and findings made throughout this research. By conducting such a detailed analysis, we can present a comparison that highlights the strengths and weaknesses of each framework.

C. Workloads

In order to test these frameworks, we had to choose models and tasks that represent diverse ML applications and tasks to ensure as comprehensive a comparison as possible. 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, and Text Summarization.

Image Classification is the task of analyzing a picture and automatically identifying and labeling the objects or subjects it contains. There are many well-known benchmarks and datasets in this field that allow us to compare performance, which provide widely accepted metrics, making it easier for comparison. For this task, we chose the ResNet50 [11] model. It is a variant of a Convolutional Neural Network (CNN), explicitly designed for image classification tasks. 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.

Automatic Speech Recognition 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. For this task we have chosen the Wav2Vec2 [12] model, specifically Wav2Vec2-base-960h, which is a model pre-trained and fine-tuned on 16kHz sampled speech audio taken from the LibriSpeech [13] dataset. 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.

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 is a sequence-to-sequence task, which can shed light on how different frameworks handle and perform with high-dimensional, textual data. The chosen BART [14] model is designed for several NLP tasks, such as text generation, translation, and summarization. It functions by first

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

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. This model tackles the challenges of language understanding, context preservation, and summarization in the broad field of NLP.

It is important to mention that 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.

D. Benchmarking Environment

After downloading and saving the model files in their respective formats, we have to create our load-testing solution. Locust was our 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 resources due to our use of High Performance Computing (HPC). All experiments were conducted on a NVIDIA Quadro RTX 5000.

After choosing our tasks, 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. We have separate containers for each framework, where we bind the respective models when it is time to test them. All of the containers have their own configuration, which are easily reproducible due to the Singularity definition files. Every experiment started with both containers being deployed using SLURM batch files. To gather data from the experiments, we ran each test 5 times, saved the results in a csv file, and took the average of all runs. We chose this to increase the reliability and reproducibility of the results, minimizing the impact of outliers and background noise. After this step was complete, we raised the number of virtual users created by our load-testing container, and repeated the previous steps, until the experiment for a single model was complete.

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, strengthening the conclusions drawn from the study.

IV. Results

All in all, the range of the recorded 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 user, 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.

A. ResNet50

For ResNet50, PyTorch consistently outperformed other frameworks in terms of throughput across all user counts. That being said, Figure 2 shows that Triton's (PyTorch) performance is trailing not too far behind, whereas TensorFlow Serving is in this case definitively slower.

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 (Requests per Second) 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, PyTorch exhibited a higher comparative performance when considering latency, followed

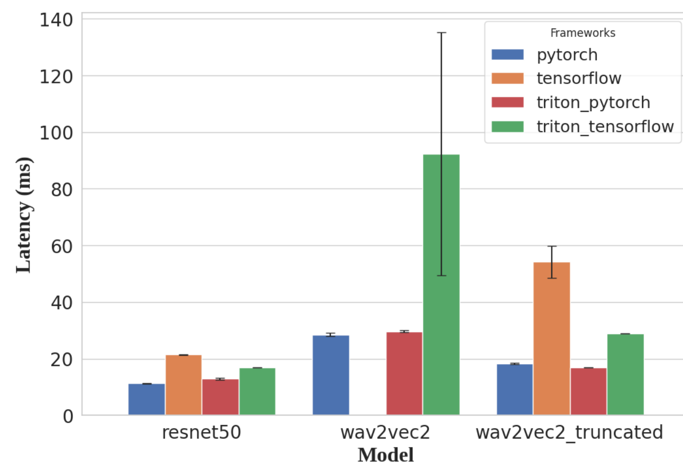

Figure 1. Latency results for 1 virtual user (in milliseconds)

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

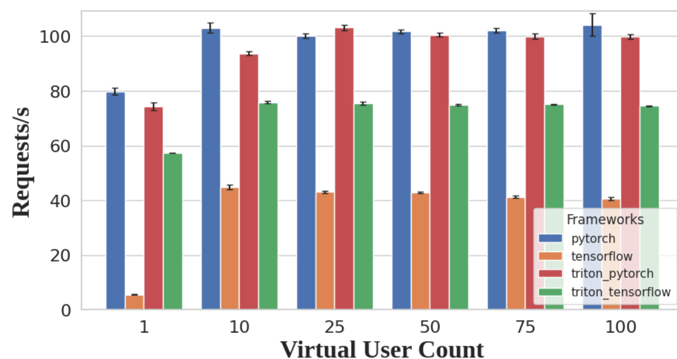

Figure 2. Throughput of the ResNet50 model (in requests per second)

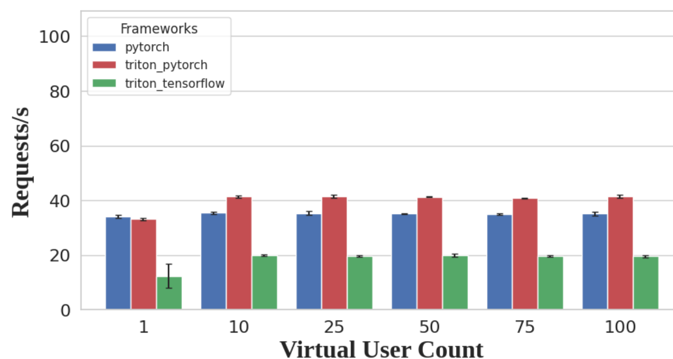

Figure 3. Throughput of the Wav2vec2 model (in requests per second)

closely once again by Triton (PyTorch). TensorFlow showed the highest latency of all the frameworks. The same relationship between the frameworks can be seen in the test with 100 users, although the numbers change slightly in user counts 10-75.

These latency results, 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 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, which will become evident later, when we examine the other models.

B. Wav2Vec2.0

Earlier on, we imposed the usability constraint that the model should be taken as-is, from official resources of the frameworks. Due to this, 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 3 that Triton and PyTorch displayed comparable performance at lower user counts, but Triton excelled as the load increased. In the test with 10 and more virtual users, all three frameworks' throughputs stay constant, implying that this is caused by the models themselves, and not by the user count.

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 20 requests at all times 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.

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 Automatic Speech Recognition (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 real scenarios, the platform is not going to constantly be under the heaviest load.

C. Wav2Vec2.0 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 to be able to continue our performance evaluation across all 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. Despite these obstacles, we believe the analysis offers interesting results into the performances of the different frameworks.

Figure 4 gives an overview of the throughput results for the Wav2Vec2 truncated model. This model exhibits the same behavior as was observed in the base Wav2Vec2 model. The truncated inputs, which are significantly shorter than the normal Wav2Vec2 model inputs, reinforce the evidence from the base Wav2Vec2 model that NVIDIA Triton provides much better performance. As before, we can see here the throughput plateauing in the tests with more than 10 users. This time, however, due to the input being significantly shorter, the absolute 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 thus far.

When considering the throughput values for each framework individually, we can see that the values are 1.5 to 2 times

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

TABLE I. FRAMEWORKS' PERFORMANCE, BASED ON THROUGHPUT AT 100 USERS (IN RPS)

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	**1.44**	-	-	-

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

D. Issues and Challenges

Before we evaluate the results of our experiments, we have to bring up the challenges faced throughout. As mentioned above, TensorFlow could not serve Wav2Vec2 due to constraints in TensorFlow Hub[15]. The SavedModel's serving signature requires an input tensor of size `(-1, 50000)`, meaning audio sequences must have 50,000 samples. Our attempts to define a custom input format failed, possibly due to a specific operation in the TensorFlow graph, specifically in the convolutional layer in the positional convolutional embedding `(pos_conv_embed)` of the Wav2Vec2 encoder.

After multiple failures, we decided to align all frameworks with the TensorFlow model's constraints. To match the required input size, we truncate or pad the audio, which allows cross-framework performance evaluation but introduces 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 all the inputs are being truncated in the same method, we are considering this step part of the preprocessing.

We faced additional challenges with serving the BART model, which can be generalized to generative models as well. Triton requires models to be converted to TorchScript, using PyTorch's JIT compiler to optimize and interpret them at runtime. However, the BART model's `.generate()` function has dynamic operations that are difficult to trace due to variable-length loops and control flows not handled well by `torch.jit.trace`. This method of tracing essentially captures the operations executed over a single forward pass to create a static graph, hence failing to correctly trace operations with dynamic control flow.[16]

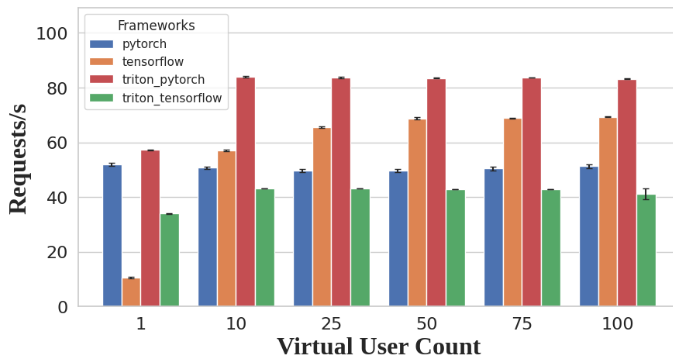

Figure 4. Throughput of the Wav2Vec2 Truncated model (in requests per second)

Serving the BART model in TensorFlow faced similar challenges due to the `.generate()` method's loops and conditionals that do not translate well to a static graph format. Issues with TensorFlow Serving arose due to dynamic tensors created by the method, which conflicts with the XLA[17]. These issues with BART highlight a serious limitation: the static graph requirements of these frameworks make it hard to handle models with complex control flows, such as generative models. We could only serve BART with PyTorch, as TorchServe does not require JIT compilation and allows custom model handlers to call the `.generate()` method during handling.

V. DISCUSSION | EVALUTION

A. Evaluation

Judging from the individual model graphs as well as Figure 1, we can see that PyTorch is overall the best when measuring prediction latency. From the same figure we can also tell that the range of the latency results for all (but one) frameworks was quite stable, thus proving that the results are accurate.

Regarding the throughput, the results were consistent across different user counts, with PyTorch and Triton (PyTorch) competing closely for first place. The throughput performance in Table I also points to the fact that the performance of TensorFlow and Triton (TensorFlow) is consistently the worst, in most cases performing around 0.5x 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.

Since PyTorch consistently shows the lowest latency in the chosen models, it is ideal for real-time applications. Whereas PyTorch excels in low latency, Triton (PyTorch) stands out for high throughput across all models, only trailing behind the former framework.

On the usability side, 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 III.

The scores shown in Table II are crucial for adressing the question of which is the most usable framework. PyTorch

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

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.

B. Discussion

Our results suggest that, although there have been quite a few differences from the Biano-AI research [18], the best-performing framework is still PyTorch. Nevertheless, the failure rate in our research was completely different from the preceding one. None of the tests we conducted showed any failed requests across all frameworks and models tested, marking a significant difference from the previous study.

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 Table 1, it is evident that the performance of the ML inference frameworks varies significantly, with each framework exhibiting strengths and weaknesses in different areas.

The lower latency PyTorch consistently demonstrates makes it 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'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 analysis pertains more to the relationship with the results, rather than the results proper. Let us consider the use case where we want to offer AI-as-a-Service. The important aspect would be to offer the users requiring this service 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.

Alternatively, in a situation 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 identify PyTorch as the best option here, scoring the highest in our usability scores (see Table II). 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.

VI. Conclusion and Future Work

This work 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.

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

TABLE II. Usability scores (1-5). Higher scores indicate better usability.

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

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

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 paper 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

[1] PyTorch, *Github - pytorch/serve*, https://github.com/PyTorch/serve, Accessed: 2024.11.15.

[2] TensorFlow, *Github - tensorflow/serving*, https://github.com/tensorflow/serving/, Accessed: 2024.11.15.

[3] Triton Inference Server, *Github - triton-inference-server/server*, https://github.com/triton-inference-server/server/, Accessed: 2024.11.15.

[4] V. J. Reddi *et al.*, "Mlperf inference benchmark", *Computing Research Repository (CoRR)*, vol. abs/1911.02549, 2019. arXiv: 1911.02549.

[5] V. J. Reddi *et al.*, "The vision behind mlperf: Understanding ai inference performance", *IEEE Micro*, vol. 41, no. 3, pp. 10–18, 2021. DOI: 10.1109/MM.2021.3066343.

[6] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding", *CoRR*, vol. abs/1810.04805, 2018. arXiv: 1810.04805.

[7] A. Vaswani *et al.*, "Attention is all you need", *CoRR*, vol. abs/1706.03762, 2017. arXiv: 1706.03762.

[8] D. 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.

[9] J. Chen and X. Ran, "Deep learning with edge computing: A review", *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019. DOI: 10.1109/JPROC.2019.2921977.

[10] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges", *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–638, 2016. DOI: 10.1109/JIOT.2016.2579198.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", *CoRR*, vol. abs/1512.03385, 2015. arXiv: 1512.03385.

[12] A. Baevski, H. Zhou, A. Mohamed, and M. Auli, "Wav2vec 2.0: A framework for self-supervised learning of speech representations", *CoRR*, vol. abs/2006.11477, 2020. arXiv: 2006.11477.

[13] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "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.

[14] M. Lewis *et al.*, "BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension", *CoRR*, vol. abs/1910.13461, 2019. arXiv: 1910.13461.

[15] Kaggle, *Wav2vec2 model on tensorflow2*, https://www.kaggle.com/models/kaggle/wav2vec2/tensorFlow2/960h/1?tfhub-redirect=true, Accessed: 2024.11.15.

[16] PyTorch, *Torch.jit.trace – pytorch 2.4 documentation*, https://pytorch.org/docs/stable/generated/torch.jit.trace.html, Accessed: 2024.11.15.

[17] OpenXLA Project, *Openxla project*, https://openxla.org/xla, Accessed: 2024.11.15.

[18] Biano AI, *Quantitative comparison of serving platforms for neural networks*, https://biano-ai.github.io/research/2021/08/16/quantitative-comparison-of-serving-platforms-for-neural-networks.html, Accessed: 2024.11.15.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org