Case study of WebAssembly Runtimes for AI Applications on the Edge

Saif eddine Khelifa*, Miloud Bagaa*, Ahmed Ouameur Messaoud*, Adlen Ksentini§
* Department of Electrical and Computer Engineering, Université du Québec à Trois-Rivières,
Trois-Rivières, QC, Canada. Emails: saif.eddine.khelifa, miloud.bagaa, messaoud.Ahmed.Ouameur@uqtr.ca
§ EURECOM, Sophia-Antipolis, France. Email: adlen.ksentini@eurecom.fr

***Abstract*—In the realm of Artificial Intelligence (AI), the need for immediate response times has given rise to the Cloud Edge Computing Continuum (CECC). This new paradigm, aided by emerging technologies, addresses latency and network delays while promoting portability, security, and efficiency, thereby enhancing Quality of Service (QoS). A noteworthy technology in this context is WebAssembly (Wasm), originally conceived to amplify web performance. It has transitioned to the CECC, primarily due to key enablers like the WebAssembly System Interface (Wasi) and the Wasm runtime. Besides offering heightened security through its sandboxing mechanism, WebAssembly's compact code paves the way for rapid cold start times and seamless migration in AI applications. However, with WebAssembly's nascent integration into the CECC, several questions arise. Prominent among them is the efficiency of deploying AI tasks in Wasm binary format, particularly the performance of Wasm runtimes in AI-centric tasks and potential factors affecting such executions. Addressing these queries, our study examines various deep-learning models on standalone WebAssembly runtimes. Our findings indicate that, for smaller networks with optimized parameters, standalone runtimes approach native performance, presenting just a 1.1x overhead on average. Contrarily, networks with an extensive parameter set exhibited pronounced overheads. We also identified multiple factors, associated both with runtimes and neural networks, offering insights for future research endeavors.**

I. Introduction

During the last decade, cloud computing gained much attention among researchers and industry; thus, cloud services have become increasingly common. Despite all of this, cloud services have become typically unable to respond in real-time to intensive computer vision tasks due to inherent service latency and network delay brought on by congestion or distance [1]. WebAssembly known as Wasm [2], is a low-level binary instruction format designed in 2015 to be used in the browser to permit security and meet portability and performance features as an extension to JavaScript performance drawbacks. Furthermore, Wasm enables the execution of the code on the web, which would be quicker and more effective. Wasm features pushed the research community to bring it beyond the web, more specifically *Edge computing*, executing Wasm in such an environment was associated with standalone runtimes also a WebAssembly System Interface (Wasi) in order to access and perform on the underlying host platform [3].

Despite the evident advantages of WebAssembly, particularly its adaptability to edge environments for enhanced security and portability, a pertinent research challenge emerges the efficiency of Wasm runtimes in deploying AI applications as Wasm binaries. The central questions of this inquiry include:

How does a standalone WebAssembly runtime perform when executing AI tasks? What quantity of memory is requisitioned by the Wasm runtime to accomplish such operations? Are all WebAssembly runtimes aptly equipped for AI tasks? The urgency in addressing these questions stems from the imperative nature of constructing portable AI solutions tailored for edge computing. When we deploy resource-intensive AI tasks without meticulous planning and optimization, we risk sub-optimal performance. Moreover, adapting the Wasm runtime to every unique system architecture is a formidable task. Given these complexities, an in-depth study gauging the efficiency of WebAssembly runtimes for AI applications becomes vital for future decision-making.

This paper aims to contribute with a use case study on AI tasks for WebAssembly standalone runtimes using Wasi with pure WebAssembly because an existing Wasi-Like solution for AI inference was already introduced called *Wasi-nn* [4]. However, this technology was not fully generalized on all WebAssembly runtimes; thus, we conducted our study on pure WebAssembly inference using Wasi to examine as many as possible runtimes with the five most used runtimes in the market today : Wasmtime [5] , Wavm [6],Wasmer [7], Wasmedge [8], Wamr [9]. Hence, we constructed a simple Case Study pipeline, that can be extended by future research to have an AI benchmark suite as MlPerf [10] benchmark developed by NVIDIA. Hence, we selected a bunch of popular Deep learning architectures: VGG [11], Mobilenet V1 [12], Mobilenet V2 [13], YuNet [14] with different parameters. In summary, the contributions of the paper are manifold, including:

- We examined the performance overhead of WebAssembly standalone runtimes for Deep learning tasks despite its portability and security.
- We created a Case Study pipeline for Wasm standalone runtimes that targets future insights on constructing an entirely pure functional WebAssembly MlPerf suite.
- We summarized our findings in the well-structured discussion that can be used to assess the choice of WebAssembly standalone runtime regarding the AI task.

The rest of the paper is organized as follows. Section II presents the background knowledge of WebAssembly and related works. Section III describes the methodology of our study. Section IV leverages the experiments and results we had. Finally, the paper is concluded in Section V.

II. Background and related works

This section explores the WebAssembly (Wasm) format, its standalone runtimes, and its current limitations in edge computing and related work upon all of this.

A. WebAssembly enablers on the edge

The WebAssembly System Interface (Wasi) is a crucial enabler for edge computing environments, providing security

by allowing access to resources, such as sockets and files. It also provides 46 functions to enable WebAssembly programs to interact seamlessly with the host platform and can be converted from POSIX to Wasi calls using popular language compilers like C and Rust [15].

The second key enabler for WebAssembly independency is the standalone WebAssembly runtimes, these runtimes can execute the WebAssembly module in two ways Interpreter mode, and JIT compilation mode both these modes are crucial for AI tasks or any applications compiled down to Wasm binaries.

Just-in-Time Compilation : JIT compilation automatically converts Wasm code to native binary code, allowing it to be performed immediately on the host computer. By exploiting the speed of compiled native code, this strategy provides significant performance advantages. JIT compilation, on the other hand, may incur a one-time compilation overhead that can be partially amortized if the produced code is executed several times [16].

Interpretation: Interpretation is a technique to read and execute WebAssembly instructions. It relies on high-level language functions. While this method is more portable and has lower compilation costs, it generally performs less than JIT compilation due to the overhead associated with interpreting instructions during execution.

Furthermore, standalone WebAssembly runtimes often mitigate the drawbacks of JIT compilation runtime by using ahead-of-time compilation (AOT) that compiles code into native binary code in advance, then loads this code into the memory whenever the Wasm module is invoked for execution; thus, it enhances the overall performances in terms of compilation time and execution time [16].

B. Wabassembly Pitfalls on Edge

- Compilation Challenges and Platform Support: Considering edge computing's heterogeneous nature and due to Wasi's limited support for system calls, compiling applications to Wasm can be problematic. Extending Wasi to conform to POSIX completely may limit Wasm's ability to execute in the many contexts prevalent in edge computing scenarios. As a result, developers must choose between portability and compatibility [15]. For instance, Wasi-nn is an extension of the Wasi-like interface to execute machine learning. Its main goal is to make an API on top of WebAssembly runtime that can access GPU hardware to perform inference of an AI task [4]. However, this can limit the portability of Wasm application due to the fact some Wasm runtimes don't implement these features, e.g. Wasmer. Thus a device that uses Wasmer as its interpreter can struggle to execute the Wasm binaries generated using Wasi-nn Wasmtime bindings; in addition, the current Wasi-nn implementation on Wasmtime runtime supports only OPENVINO as backend, which by itself supports only Intel processors, thus limits further the portability aspect even further. Other recent works were to adapt Wasi-nn to onnx through onnx-runtime [17]. However, there is still a long way to go for fully portable API AI inference.
- Memory Limitations: Edge devices often have limited resources, such as memory capacity. Wasm presents an issue of memory footprint [15], which may not be enough for some edge computing use cases. Borui Li et al. have proposed a solution to this problem [18], whereby a lightweight WebAssembly runtime has been suggested for resource-constrained devices.

C. Related work

We discuss how prior initiatives inspired and encouraged us in this part. Even though state-of-the-art [16], [19] described a few AI tasks, without mentioning whether deep learning networks were employed or if a machine learning algorithm was used. Furthermore, they ran their tests on a single image input without considering batch size, which is an essential factor in AI inference. Finally, they rely on their research on the overall process time of the AI task, which includes both loading and inference time. This distinction is vital because loading time can be improved by using better-optimized Wasi interfaces I/O. In contrast, inference time is a parameter related to both runtime and the AI task, i.e., the neural network. Our work varies from others in that our goal is to investigate the effectiveness of WebAssembly runtimes for AI tasks carefully and the feasibility of running these tasks on independent runtimes to meet the requirements of the edge computing environment, i.e., AI as a Service (AIaaS). Finally, our study introduces a simple case study pipeline, which is most likely a simplified version of MlPerf.

III. Methodologies

In this section, we describe our process throughout this research. First, we conducted our study by looking at the candidate deep learning networks that work with WebAssembly runtimes, followed by runtime selection, and finally, we established a case study pipeline and characterized our results.

A. Model selection

Considering the constraints imposed by the *tract_tensorflow* library in Rust, we carried out a series of experiments on various computer vision tasks and networks that were publicly accessible. From the plethora of pre-trained models at our disposal [20], we selected those most commonly employed for classification and object detection tasks, which were highly compatible with *tract_tensorflow*. Nevertheless, it was necessary to make some changes to the source code. The fixed issues have been pushed to the tract public GIT repository. The models that operated successfully after these adjustments are described in Table I

B. Runtimes selection

To select WebAssembly runtimes, we based on the following criteria :

- Support of Wasi is a must since we are going to use file system I/O to load the models
- Toolchain maturity covers and runs on most of Wasi instructions and functions
- Runtime is recently underdeveloped and being used by the community
- Runtime that handles AI models inference *(example : Wasm3 [21] can't handle AI tasks)*

These criteria are crucial for both the technical and the research parts; thus, the selected runtimes are the following:

- **Wavm**: [6] is a standalone WebAssembly runtime that executes WebAssembly programs using an LLVM-based JIT compiler. It converts WebAssembly code to LLVM Intermediate Representation (IR), then uses an LLVM

JIT compiler to convert LLVM IR to native binary code. Wavm can produce high-quality native binary code by exploiting the numerous optimizations within the LLVM architecture.

- **Wasmer**: [7] Wasmer is a WebAssembly runtime that can function as a standalone or library embedded in languages. It supports JIT compilers, such as SinglePass, Cranelift, and LLVM, offering a trade-off between compilation speed and native code quality.
- **Wasmedge**: [8] Wasmedge is a lightweight, high-performance, and extensible WebAssembly runtime for cloud-natives provides a well-defined execution sandbox that uses an interpretation-based execution mechanism.
- **Wasmtime**: [5] Wasmtime is the official WebAssembly runtime developed by the Bytecode Alliance. It employs a Just-In-Time (JIT) compilation based on Cranelift to execute WebAssembly binaries.
- **Wamr** : [9] The WebAssembly Micro Runtime (Wamr) is a lightweight, standalone WebAssembly runtime built for low-power, resource-constrained devices. it uses an interpretation-based execution mechanism.

C. Case study pipeline

In our research, we carried out a pipeline that began with extracting checkpoints from pre-trained models. Subsequently, we constructed a custom freezing function utilizing the TensorFlow v1 library within TensorFlow v2 [22], as TensorFlow v1 is no longer supported. This freezing function allowed us to convert the checkpoints into a frozen graph (protobuf) format compatible with the tract_tensorflow library. Following this, we loaded the model into Rust and employed the cargo and Wasm32-Wasi toolchain for further processing and execution.

IV. Performance and Results

In this section, we study the generated results from our case study and test the performance of our WebAssembly standalone runtimes on three metrics: *Memory* against deep-learning models parameters, *Execution time* against deep-learning models parameters, and *Inference time-Load time*. We have also studied the optimized binaries using the AOT (ahead of time) method. The case study was tested on Hardware specifications as follows: Processor AMD architecture x86 Ryzen 9 5900X CPU and 12GB RAM DDR4 main memory, The operating system Ubuntu 20.04 LLVM 12.0, Cargo 1.69 It is imperative to highlight that our research endeavors to delineate and categorize the variables involved in executing artificial intelligence tasks in WebAssembly (Wasm) binary format. This is undertaken to guide future research in the field, Moreover, as mentioned above, all test codes were performed in default 0-level optimization all models were tested on the publicly available dataset ImageNet [23] since it is the most used dataset for different computer vision tasks with 10000 classes. Finally, it's noteworthy that some WebAssembly runtimes failed to execute some of the models due to memory limits like Wamr and great time overhead like Wasmedge thus some combinations may not be found in the graphs, Also, all values in all graphs are modified through the equation below 1 for clear graphs visualization value **'v1'** can be **'GB'** memory or **'Seconds'** for time and **"v2"** is the displayed value that results from the log operation, we added **'2'** since some values can be negative :

$$v2 = log10(v1) + 2 \tag{1}$$

Finally, the deep learning models are orchestrated from the smallest to the largest network based on the number of parameters.

A. Overall execution Performance

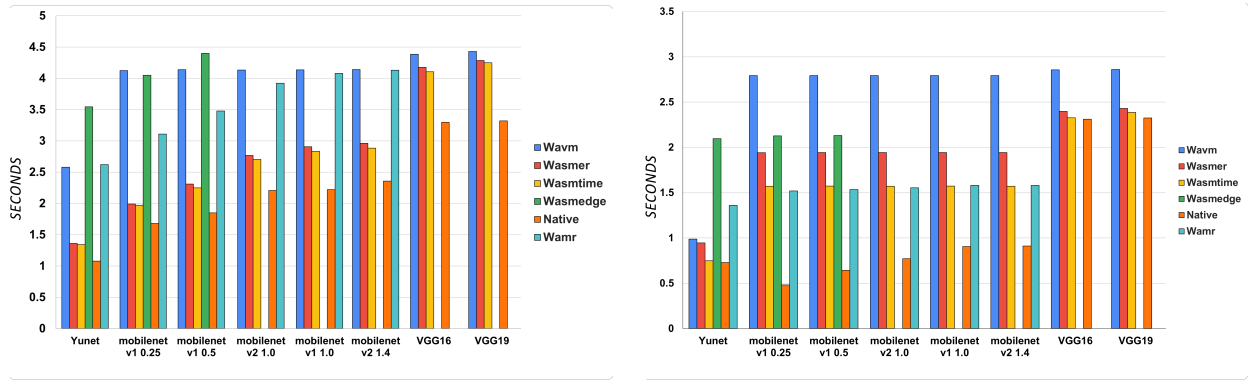
In our study, we have conducted experiments on various models utilizing Level 0 optimization compilation without any pre-compilation. The results are presented in Figure 1(a), where the native runtime serves as the baseline for comparison. As shown in Figure 1(a), the overall WebAssembly runtimes have different time execution results that gradually amplified from the smaller networks moving to larger networks with extensive parameters, We have observed several notable aspects upon examining the results in Figure 1(a). First and foremost, Wasmtime and Wasmer outperformed other runtimes with a 10x difference of speed on average in all tests (in real values), primarily due to their Just-In-Time (JIT) compilation nature. JIT compilation is known to generate higher-quality code, a fact that is well-established in the state of the art [19].In contrast, the poorest results were generated from the part of Wasmedge due to the interpretation feature, however, an interesting note is that Wavm and Wasmer, Wasmtime use JIT compilation, still, Wavm introduces a huge overhead compared to Wasmer and Wasmtime due to the way these runtimes translate the code as it's stated in the [19] [6] [7] [5] Wasmtime and Wasmer translate the code into Cranelift IR then to a machine code, In contrast, Wavm translates the code into LLVM IR then into machine code, Because LLVM optimization passes and strategies are mature and have been refined over many years, Still was outperformed by runtimes that use Cranelift which its main goal was to quickly translate WebAssembly and other IRs to machine code with decent performance, Overall, we deduced that code translation time to Intermediate format (Cranlefit IR /LLVM IR) plays a crucial role in executing AI applications.

Another fascinating observation pertains to the network parameters of the AI models. We found that for most MobileNet networks with varying convergence parameters, there was only a small performance gap between Wasmtime, Wasmer, and native runtimes. However, this gap widened considerably over 10x slower on average than native (in real values) due to the number of model parameters increasing from tens of millions to hundreds of millions of parameters, adversely impacting all runtimes. Finally, some runtimes encountered failures with certain networks. For instance, Wasmedge struggled with larger networks due to prolonged execution times. On the other hand, Wamr failed with the VGG's network because of memory constraints.

This implies that an optimized neural network tailored for mobile devices and IoT devices could be a suitable use case for AI as a service in the WebAssembly format. Furthermore, our findings suggest that, in the context of AI tasks, interpretation-based compilation still has a long way to go before it can surpass the performance of JIT compilation. In conclusion, WebAssembly runtimes standalone can be helpful in terms of execution when it comes to serving optimized neural networks for edge computing continuum environments using the already mentioned methods in [24].

TABLE I: Table shows the selected models and their respective details

	Neural network	nn Parameters	input Parameters	Other Parameters
1	VGG-16	138 M	1 input image and batch-size of 3	16 conv layers
2	VGG-19	143.7 M	1 input image and batch-size of 3	19 conv layers
3	MobileNet V1_1	4.24 M	1 input image and batch-size of 3	Width multiplier = 1
4	MobileNet V1_0,5	1.34M	1 input image and batch-size of 3	Width multiplier = 0,5
5	MobileNet V1_0,25	0.42M	1 input image and batch-size of 3	Width multiplier = 0,25
6	MobileNet V2_1	3.47 M	1 input image and batch-size of 3	Width multiplier = 1
7	MobileNet V2_1,4	6.5 M	1 input image and batch-size of 3	Width multiplier = 1,4
8	Yunet	75 856	1 input image	



(a) Graph shows the Time overall using equation 1 execution for each runtime to the respective networke

(b) Graph shows the Memory consumption known as Maximum resident set size for each runtime to the respective network using equation 1

Fig. 1: figure shows the Time and Memory results against different deep learning networks (from smaller to larger)

B. Memory Overhead

In this section, we continue our study on memory, as previously mentioned, the memory footprint is a significant limitation of WebAssembly technology. Researchers have already begun addressing this issue. Standalone runtimes typically attempt to load models into memory using Wasi interfaces, resulting in substantial memory consumption. It is crucial to examine the key factors contributing to this drawback.

During our experiments, we measured gigabytes (GB) of memory consumption, as some runtimes exhibited high memory usage for some networks *("VGG")*. We extracted our results by determining the maximum resident set size (MRSS), which reflects the peak memory consumption during a process's execution.

Upon analyzing Figure 1(b), we observed that Runtimes with JIT compilation have consumed less memory compared to others over x5 less memory on average. Interestingly, Wamr did not consume as much memory as Wasmer, despite the latter's superior performance in terms of time. Upon further investigation, we discovered that Wamr has a fixed stack size value, which limited memory usage for such tasks. Although the stack size was set to the maximum during our case study, as shown in Figure 1(b), Wamr's memory consumption was notably lower than that of Wasmer. This is because Wamr runtime is specifically designed for IoT devices and other resource-constrained devices, leading to more efficient memory management. [9].

Another noteworthy observation is that memory consumption remained relatively stable across all runtimes for the MobileNet networks. This can be attributed to two factors: The converged number of parameters, which determines the memory allocated for computing these parameters during the feed-forward pass, and the similar file sizes of the MobileNet networks, which affect the amount of memory needed to load the models from disk into WebAssembly's linear memory.

Finally, Wasmer and Wasmtime demonstrated near-native memory consumption, even when dealing with large networks, such as VGG, which have hundreds of millions of parameters, resulting in memory consumption of around 2.5 GB for each. These results highlight the potential for deploying neural network models in edge computing environments using JIT compiler standalone runtimes, as both Wasmtime and Wasmer effectively minimized memory usage with negligible performance overhead in larger networks compared to native.

C. AOT Compilation performance

In our pursuit of investigating WebAssembly standalone runtimes execution, we conducted a fair comparison with native performance as the baseline by examining the overall execution time of all tasks and runtimes that support Ahead of Time (AOT) compilation as shown in Figure 2 while excluding Wamr due to its inefficiency to provide AOT compilation format. As anticipated, utilizing AOT compilation led to a substantial increase in execution speed for almost all runtimes. The following improvements were observed: Wavm's execution time improved by 75% of its original duration, significantly enhancing its performance and surpassing other runtimes during the VGG networks execution. This improvement can be attributed to the mature optimization capabilities of the LLVM toolchain that Wavm employs to translate code into LLVM IR for execution. Moreover, Wasmedge interpreter-based runtime outperformed others in Yunet and MobileNet tasks but did not perform as well in tasks with higher numbers of parameters. Wasmedge's execution time was optimized by 6.57% of its original value.

Furthermore, JIT compiler-based runtimes, namely Wasmtime and Wasmer, saw optimizations of 10% and 16% from their original performance metrics, respectively. Despite these improvements, they introduced some overhead in smaller networks against interpreter-based runtimes. We conclude that for AI tasks, particularly on smaller networks, an interpreter compiler combined with AOT (Ahead-of-Time) compilation is notably more effective than JIT compilers. The LLVM

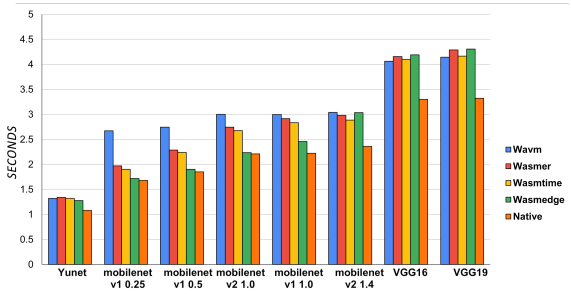


Fig. 2: Graph shows the Time overall execution in AOT mode for each runtime to the respective network using equation 1

toolchain offers significant potential for code optimization on larger networks. However, attaining near-native performance for these bigger networks still poses a significant challenge.

D. Inference and load time

As we proceed to examine the execution time for our neural networks, it is crucial to consider the inference time, as it is a key factor in the overall execution time, which consists of both loading and inference. Analyzing these components separately can lead to significant conclusions, as they are related to different origins of problems. Inference time is associated with the neural network while loading time is connected to the Wasi interface responsible for I/O files. We have provided the percentages of inference time contribution to the overall execution in Table II. Upon further analysis, we observed that, on average, inference time does not dominate the execution process for smaller networks. However, when dealing with larger networks with many parameters, inference time becomes an essential contributor. Interestingly, this is not the case for some runtimes, such as Wasmer and Wasmtime, which use JIT compilation. We noted that the inference time in larger networks, such as VGG networks, is lower than in other networks. This can be attributed to various factors, including the relationship between bytecode execution and neural network operators. VGG and MobileNet do not necessarily include the same operators (e.g., Depthwise layer), so compiling these operators using JIT compilers can impact inference time. On the other hand, with interpreter-based runtimes *(Wasmedge)*, the inference time contributes less to the execution of the candidate neural network compared to JIT compiler-based runtimes. This suggests that improvements can potentially be made to the Wasi (loading) part to enhance performance and possibly surpass JIT compiler-based runtimes. In conclusion, carefully examining how runtimes and their compilers execute AI-related operators may be the key to developing AI-friendly runtimes that provide secure, portable, and more efficient AI services.

E. Batch impact on execution time and memory

Finally, we examine our results in a more advanced scope with a very important factor in the AI field, which is the batch size of inputs. We did not want to overload the WebAssembly runtime and thus take a huge time to execute so we fixed the batch size of 3 images from imagenet dataset with the dimension of 224x224 and 3 channels. We have conducted the examination, and we explored both memory and execution time using the equation (2), whereby t1 is the time for a neural network for specific runtime with batch mode and t2 is the time for a neural network for specific runtime without batch mode. We have averaged all neural network execution times for a specific runtime for both modes.

$$avg(t1) \div (avg(t2)) \tag{2}$$

Examining Table III, We observed longer execution times in most JIT compilers, except for Wavm, which uses an LLVM IR-based JIT. While JIT compilers generally outperform interpreter compilers in batch execution without AOT mode, it's plausible that interpreter compiler-based runtimes like Wasmedge might excel for larger networks than VGG, given their minimal time and memory increments in batch mode. This area merits further investigation.

Moving on to memory overhead, we observe that all of the runtimes, including native, require nearly the same amplification in the amount of memory. In conclusion, increasing the batch size has an unfavorable effect on both time and memory for runtimes. Batch data is commonly associated with the SIMD (single instruction multiple data) feature, which can considerably minimize this time and memory expense, While WebAssembly currently doesn't support this feature our investigations did not consider this feature even for the native. Finally, the batch is a crucial factor when deploying AI tasks in the WebAssembly binary format. To ensure maximum efficiency, the batch must be fine-tuned according to the target device or platform.

F. Discussion

In this section, we will summarize and discuss our interesting findings during the examination, which will be categorized as follows:

- **Important parameterization during AI service deployment**: As discussed in previous sections and according to our findings deploying AI services will depend on three factors at least, target compilation for runtime JIT or interpreter mixed with a head of time, the second factor is Parameters of neural network that is a very crucial factor and need to be considered for optimization before running any AI task on WebAssembly runtimes. Hence, there are many ways to optimize neural networks, such as pruning, quantization, and knowledge distillation [24], also another way of optimization is using level 2 optimization or level 3 optimization; the last and most important factor is the batch size of images depending on the target platform configuration taking consideration of this factors can impact the deployment of AI services in Wasm binary format greatly, thus we can benefit from the portability and security without losing much of efficiency.

- **Runtime for which task**:
  Selecting a runtime for a certain task can be tricky, and based on our findings JIT compilers like Wasmtime and Wasmer did a great job overall, however, Wamr consumes less memory in most of the networks, but it couldn't execute huge graphs due to its limited stack size so Wamr can a good choice for resource-constrained devices with an average performance However if it's not the case Wasmtime and Wasmer with based JIT compilation can deliver high near-native performance on a medium, smaller network, however, other runtimes like Wasmedge

TABLE II: Figure shows the inference time contribution to the execution time

	Wasmtime	native	Wamr	Wavm	Wasmer	Wasmedge
mobilenet_v1_0.25_128	39.55%	20.55%	40.89%	38.98%	42.42%	44.75%
mobilenet_v1_0.5_160	68.20%	32.50%	68.94%	64.63%	96.04%	70.06%
mobilenet_v2_1.0_224	97.91%	54.97%	83.22%	97.57%	98.12%	failed
mobilenet_v1_1.0_224	98.59%	56.47%	88.17%	98.09%	98.65%	failed
mobilenet_v2_1.4_224	98.00%	92.36%	86.68%	83.35%	87.33%	failed
Yunet	51.16%	85,7%	50,61%	59.72%	50%	50.39%
VGG16	86.25%	5.90%	failed	86.25%	80.60%	failed
VGG19	88.90%	5.87%	failed	88.90%	83.75%	failed

TABLE III: Table represents the impact of batch size on execution time and memory

Time					
Wasmer	Wavm	Wasmtime	Wamr	Wasmedge	native
3x	1,38x	2,8x	2,75x	2,25x	1,23x
Memory					
Wasmer	Wavm	Wasmtime	Wamr	Wasmedge	native
1,14x	1,2x	1,3x	1,25x	1,22x	1,24x

based on interpreter can bypass this feature on a medium smaller network using a head of time compilation, In contrast, it may encounter a compilation overhead. Finally, WebAssembly runtimes have a long way to go compared to the native performance on AI tasks; thus it is better to make sure of the necessity of having portability and security issues for certain scenarios since native could be the best choice after all

- **Customization, SIMD Features, and AI Applications on WebAssembly**: To consolidate our findings, WebAssembly runtimes are not always suboptimal at executing inference code. This might be related to loading delays or how AI operators are produced using JIT compilers for various runtimes. Customizing the compilation process, coupled with efforts to minimize loading times for such models, can significantly enhance WebAssembly runtime performance. Additionally, our observations on batch size underscore the importance of incorporating SIMD features and GPU support, thus generalizing Wasi-nn can unlock these features for wasm modules. While Wasi-nn has a profound impact on WebAssembly runtime efficiency, designing and creating WebAssembly inference purely without Wasi-nn allows for heightened customization and better control. Deploying AI applications on WebAssembly, given the current state, demands meticulous neural network optimizations like pruning, quantization, and selecting appropriate optimization levels. The choice of runtime, whether JIT compilers such as Wasmtime and Wasmer for medium to smaller networks or interpreter-based solutions like Wasmedge with AOT mode for larger ones or the choice of having batched inference or not, is crucial. Inherent trade-offs, such as memory limit in Wamr or potential compilation overheads with Wasmedge, should be noted. In essence, continued research should investigate the tailored optimization of WebAssembly runtimes for AI tasks, with an emphasis on integrating SIMD features and GPU support.

V. Conclusion

In this paper, we investigated everything from memory to time execution for standalone WebAssembly runtimes to construct a simple case study and study the factors related to deploying AI services on edge computing, resulting in exciting findings. The obtained results not only contribute to the community but also push further research and investigation to be done and the developer community to look more in-depth into their developed technologies; Why not we achieve the goal of having Portable, Secure, Efficient AI services across edge computing environment by leveraging WebAssembly technology.

References

[1] 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. 637–646, 2016.

[2] Webassembly. [Online]. Available: https://webassembly.org/

[3] Wasi. [Online]. Available: https://wasi.dev/

[4] Wasi-nn. [Online]. Available: https://github.com/WebAssembly/wasi-nn

[5] Wasmtime. [Online]. Available: https://github.com/bytecodealliance/wasmtime

[6] Wavm. [Online]. Available: https://github.com/WAVM/WAVM

[7] Wasmer. [Online]. Available: https://github.com/wasmerio/wasmer

[8] Wasmedge. [Online]. Available: https://github.com/WasmEdge/WasmEdge

[9] Wamer. [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime

[10] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "Mlperf inference benchmark," pp. 446–459, 2020.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv 1409.1556*, 09 2014.

[12] A. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 04 2017.

[13] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," pp. 4510–4520, 06 2018.

[14] W. Wu, H. Peng, and S. Yu, "Yunet: A tiny millisecond-level face detector," *Machine Intelligence Research*, 04 2023.

[15] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Webassembly as a common layer for the cloud-edge continuum." New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3526059.3533618

[16] W. Wang, "How far we've come – a characterization study of standalone webassembly runtimes," pp. 228–241, nov 2022. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IISWC55918.2022.00028

[17] Wasi-nn-onnx. [Online]. Available: https://github.com/deislabs/wasi-nn-onnx

[18] B. Li, H. Fan, Y. Gao, and W. Dong, "Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, ser. MobiSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 261–272. [Online]. Available: https://doi.org/10.1145/3498361.3538922

[19] Z. Wang, J. Wang, Z. Wang, and Y. Hu, "Characterization and implication of edge webassembly runtimes," pp. 71–80, 2021.

[20] Tensorflow slim pretrained models. [Online]. Available: https://github.com/tensorflow/models/tree/master/research/slim

[21] Wasm3. [Online]. Available: https://github.com/wasm3/wasm3

[22] tensorflow. [Online]. Available: https://www.tensorflow.org/

[23] Imagenet. [Online]. Available: https://www.image-net.org/index.php

[24] M. M. H. Shuvo, S. K. Islam, J. Cheng, and B. I. Morshed, "Efficient acceleration of deep learning inference on resource-constrained edge devices: A review," *Proceedings of the IEEE*, vol. 111, no. 1, pp. 42–91, 2023.