# Anatomizing Deep Learning Inference in Web Browsers

QIPENG WANG, Peking University, Beijing, China
SHIQI JIANG, Microsoft Research Asia, Beijing, China
ZHENPENG CHEN, Nanyang Technological University, Singapore, Singapore
XU CAO, Microsoft Research Asia, Beijing, China
YUANCHUN LI, Institute for AI Industry Research (AIR), Tsinghua University, Beijing, China
AOYU LI, Nanyang Technological University, Singapore, Singapore
YUN MA, Peking University, Beijing, China
TING CAO, Microsoft Research Asia, Beijing, China
XUANZHE LIU, Peking University, Beijing, China

Web applications have increasingly adopted Deep Learning (DL) through *in-browser inference*, wherein DL inference performs directly within Web browsers. The actual performance of in-browser inference and its impacts on the Quality of Experience (*QoE*) remain unexplored, and urgently require new QoE measurements beyond traditional ones, e.g., mainly focusing on page load time. To bridge this gap, we make the first comprehensive performance measurement of in-browser inference to date. Our approach proposes new metrics to measure in-browser inference: responsiveness, smoothness, and inference accuracy. Our extensive analysis involves 9 representative DL models across Web browsers of 50 popular PC devices and 20 mobile devices. The results reveal that in-browser inference exhibits a substantial latency gap, averaging 16.9 times slower on CPU and 4.9 times slower on GPU compared to native inference on PC devices. The gap on mobile CPU and mobile GPU is 15.8 times and 7.8 times, respectively. Furthermore, we identify contributing factors to such latency gap, including underutilized hardware instruction sets, inherent overhead in the runtime environment, resource contention within the browser, and inefficiencies in software libraries and GPU abstractions. Additionally, in-browser inference imposes significant memory demands, at times exceeding 334.6 times the size of the DL models themselves, partly attributable to suboptimal memory management. We also observe that in-browser inference leads to a significant 67.2% increase in the time it takes for GUI components to render within Web browsers, significantly affecting the overall user QoE of Web applications reliant on this technology.

CCS Concepts: • **Human-centered computing** → **Empirical studies in ubiquitous and mobile computing**; • **Computing methodologies** → **Artificial intelligence**;

Additional Key Words and Phrases: Deep learning, Web browser, measurement

## 1 Introduction

**Deep Learning (DL)** has been increasingly adopted in various Web applications by deploying DL models in Web browsers for inference tasks, i.e., *in-browser inference* [6, 10, 13, 18, 45, 92]. In this context, DL frameworks and models are encapsulated and delivered within Web pages without additional installations on user devices, and DL inference is performed directly in Web browsers [92], effectively streamlining user access to DL-powered functionalities. Many Web sites have already deployed in-browser inference. For example, online Office 365 [6] utilizes in-browser inference to assist with the online editing of documents and slides; online Teams blurs the background of people for video meetings [13]; Facetest [10], Aige [18], and Theremix [8] apply in-browser inference for tasks including image classification, face recognition, and music generation.

The proliferation of DL-powered Web applications makes in-browser inference a critical topic in various research communities, including **Software Engineering (SE)** [63, 64, 74, 75, 88, 99], Web [92, 121], and Mobile Computing [80, 108]. Firstly, the performance of in-browser inference is crucial to Web-based software applications. For instance, understanding the detailed performance of in-browser inference can assist software developers in refining the applications, thereby offering users a smoother and better service [119, 122]. Moreover, the **Quality of Experience (QoE)** is pivotal in Web contexts [109], as it reflects the user's experience while browsing Web pages. Improved QoE can attract a larger user base, making it vital for the development and deployment of software applications.

Existing research efforts primarily focus on DL inference performance in native[1] [63, 64, 74, 75, 88, 106, 121], regarding inference latency, memory footprint, and so on. For instance, Tang et al. [106] accelerate DL model inference in PCs and mobile devices through a lookup table; Gao et al. [74] estimate the GPU memory footprint during DL model training in the server. However, in the scenario of in-browser inference, the browser acts as an intermediary, isolating Web applications from the underlying operating systems and restricting the application's ability to fully harness hardware capabilities. This creates a distinction between the browser environment and the native environment. The current work demonstrates an insufficient exploration of in-browser inference and there are significant differences between browser and native environments, e.g., the absence of some advanced **Single Instruction Multiple Data (SIMD)** instructions. These factors contribute to a notable knowledge gap in the literature regarding in-browser inference performance. This gap underscores the need for more focused research to better understand and optimize the performance of in-browser inference. Moreover, existing Web QoE measurement efforts primarily focus on the loading phase of Web pages such as **Page Load Time (PLT)** [28, 36, 59, 87, 107, 120]. For example, Speed Index [36] measures how quickly the Web page is visually displayed during page loading. To the best of our knowledge, no previous work has considered the impact of in-browser inference on the QoE.

To fill the knowledge gap, we present the first comprehensive measurement study of in-browser inference from two aspects: inference performance and QoE. Our extensive study involves 9

---

[1]We use "native" to refer to the non-browser environment, including servers, PCs, and mobile devices.

representative DL models across Web browsers of 50 popular PC devices and 20 mobile devices. Furthermore, we use the most widely adopted DL frameworks for deploying DL models in browsers, including **TensorFlow.js (TF.js)** [41] and **ONNX Web Runtime (ORT.js)** [31].

For *inference performance*, we consider two critical metrics: *latency* and *memory footprint*. These metrics provide insights into the time overhead and memory requirements, respectively. We present a deep analysis of model-level/kernel-level latency and memory footprint, where a kernel is the basic unit of a DL model, such as a convolution operation. Furthermore, the inherent variability in device capacities motivates us to analyze the performance variance of in-browser inference across devices.

We observe a significant disparity in prediction latency between in-browser and native inference, averaging 16.9× on CPU and 30.6× on GPU on PC devices, and the gap on mobile CPU and mobile GPU is 15.8× and 7.8×, respectively. In the CPU case, this disparity arises primarily from the underutilization of SIMD instructions on PC devices, **WebAssembly (Wasm)** runtime-inherent inefficiency, as well as competition for resources with inherent browser tasks. On the GPU side, the inefficiency of software libraries on the Web, i.e., WebGL [52], compared to native CUDA [16], contributes to the gap. Furthermore, our analysis reveals substantial performance variations across different devices, with disparities reaching up to 28.4× on CPU and 19.4× on GPU. These variations stem from the diverse capacities of devices and the hardware affinities of predesigned DL kernels. Additionally, we identify ongoing concerns regarding memory footprint during in-browser inference. In certain scenarios, the memory footprint can escalate to 334.6× the model size during super-resolution model inference. These findings underscore the inefficiency of the in-browser inference on memory management when the intermediate data size increases.

For *QoE*, we are the first to measure it within the context of in-browser inference, proposing new metrics dedicated specifically to this endeavor beyond existing QoE metrics. Specifically, we quantitatively measure in-browser inference QoE from three aspects: *responsiveness*, *smoothness*, and *inference accuracy*. Responsiveness measures the speed at which user requests are successfully responded to. Smoothness evaluates the fluidity and seamless rendering of Web page content. Inference accuracy gauges the accuracy of DL models within Web applications. Our findings reveal an interesting observation: reducing inference latency does not necessarily equate to an improvement in QoE. For instance, while WebGL-based inference generally boasts faster execution time compared to Wasm, it also results in significant degradation of up to 62.7% in smoothness due to resource conflicts arising from the competition between inference and the Web application's reliance on GPU resources.

Based on the results, we provide a series of actionable implications for various stakeholders. (1) *Web browser vendors* can extend support for Wasm SIMD and streamline its integration with in-browser inference frameworks to avert potential performance degradation. (2) *Web application developers* can enhance the QoE by configuring inference to utilize the Wasm backend, albeit with a slight increase in inference latency. (3) *DL framework vendors* can employ efficient memory management techniques, such as model information-guided memory pool, to mitigate the memory overhead during inference.

To summarize, this article makes the following contributions:

—We conducted a detailed and granular analysis of in-browser inference performance at both the model and kernel levels across multiple devices and examined the performance gap and underlying reasons compared to native inference.
—We proposed specific QoE metrics tailored for in-browser inference scenarios and conducted quantitative analyses to investigate the impact of in-browser inference on QoE.
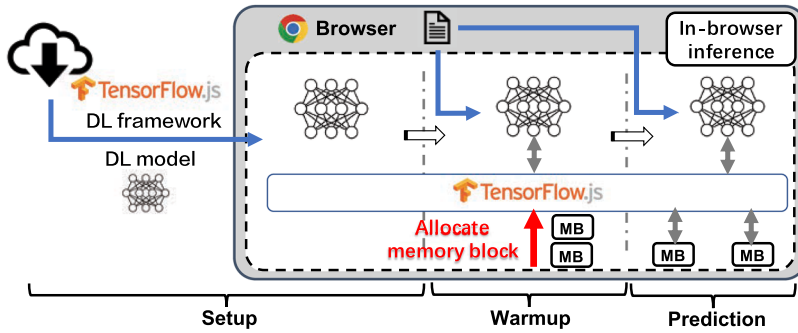
Fig. 1. Workflow of in-browser inference. "MB" denotes memory block.

—We provided recommendations for browser vendors, inference framework developers, and
   Web application developers to help them enhance the performance of in-browser inference
   and improve user QoE.
—As an additional contribution, we publicly released our code and data[2] to facilitate the replica-
   tion and extension of this work.

## 2  Background

### 2.1  In-Browser Inference

Web browsers have emerged as a pivotal platform for the widespread adoption of DL technologies
[67, 92].

Different from native applications that require installation and upgrade, raising barriers for end
users, and hindering DL service deployments, Web browsers, however, are promising platforms
for DL adoption, without the requirement for extra installation. Besides, in-browser inference
does not need to send the user data to the remote server, preserving user privacy. In-browser
inference has attracted tremendous research attention [63, 64, 71, 74, 75, 80, 82, 88, 92, 99, 108, 121].
In-browser inference empowers many new abilities of Web applications. For example, Web Office
365 integrates grammar checking and typing completion [6]; online Teams [7] uses in-browser
inference to analyze facial contours for online meetings; Facetest [10] applies in-browser inference
for face detection; Aige [18] applies in-browser inference to guess the age of users according to
face image; Theremix [8] utilizes in-browser inference for music generation according to users'
control. Many existing Web applications can benefit from in-browser inference. For instance, online
subtitle generation will be enabled for living applications; Web augmented reality, virtual reality,
and mixed reality may be more vivid when leveraging DL.

In-browser inference is the process where Web applications execute DL inference directly within
Web browsers. Figure 1 illustrates the workflow of this process, which comprises three primary
stages: setup, warmup, and prediction. During the setup stage, Web applications download the
necessary DL models and DL frameworks and then load them into Web browsers. In the warmup
stage, browsers, assisted by DL frameworks, allocate the required memory for inference and initiate
the first DL inference operation. Finally, in the prediction stage, DL models carry out subsequent
inference tasks using the capabilities provided by DL frameworks. The warmup and prediction
stages in the native environment are identical to those in the Web environment, while the setup
stage differs between the two environments. In the native environment, the setup stage merely

---

requires loading the model and framework from local storage, eliminating the need to download them from the server.

## 2.2 Frameworks and Backends

Dedicated DL frameworks are developed to facilitate DL deployment in Web applications and in-browser inference. Among them, TF.js [41] and ORT.js [31] are actively maintained and have garnered the most GitHub stars. TF.js is an offshoot of TF [56] and maintains compatibility with it, while ORT.js is derived from ORT [29, 31]. In this study, we consider both of these frameworks due to their widespread adoption for in-browser inference in real-world Web applications [67, 75, 92].

Both TF.js and ORT.js support using backends for inference acceleration, including Wasm [46, 77] on CPU and WebGL [49] on GPU. Wasm serves as a binary instruction format for a stack-based **Virtual Machine (VM)** within Web browsers. It serves the purpose of a portable compilation target for programming languages, facilitating their deployment on the Web. Wasm stands out for its efficiency, owing to its compact size and swift loading capabilities [46]. In Wasm, SIMD and multithreading are available for computation efficiency. SIMD is a hardware architecture that processes multiple data elements in one instruction, greatly enhancing computational speed. Multithreading is a technique that allows a program to execute multiple tasks concurrently, making efficient use of system resources for improving performance. WebGL is a JavaScript library designed for rendering graphics directly within Web browsers, implementing an API that adheres to the OpenGL standards [32]. Developers can harness the power of WebGL by crafting WebGL shaders [52] to expedite parallel computation processes.

Recently, a new backend called WebGPU has been introduced [53]. Nevertheless, the WebGPU standard has yet to attain official release status. WebGPU is still in its early and continually evolving phase, with no support for general models but only limited ones in TF.js. Therefore, we opt not to incorporate WebGPU into this article.

## 3 Methodology

### 3.1 Experimental Setup

We start by outlining our choices for DL frameworks, backends, DL models, PC devices, and Web browsers.

*Selection of DL Frameworks and Backends.* For in-browser inference, we opt for TF.js 4.2.0 and ORT.js 1.14.0 as our DL frameworks. Our selected backends include Wasm and WebGL, while we omit the pure JavaScript and WebGPU backends due to the lack of support for these options in ORT.js.

*Selection of DL Models.* We select nine representative DL models for study, covering three common types of Web applications, i.e., image classification, object detection, and grammar checking. For each type, we select three widely adopted DL models. Specifically, for image classification, we select MobileNetV2 (denoted as M1; 14 MB in size) [101], ResNet50 (denoted as M2; 98 MB in size) [78], and VGG16 (denoted as M3; 528 MB in size) [103]; for object detection, we select SSD-MobileNetV2 (denoted as M4; 26 MB in size) [89], EfficientDet (denoted as M5; 23 MB in size) [105], and Yolo5-Middle (denoted as M6; 83 MB in size) [100]; for grammar checking, we select Bert-base (denoted as M7; 104 MB in size) [57], MobileBert (denoted as M8; 95 MB in size) [104], and Bert-Small (denoted as M9; 17 MB in size) [70]. We use the pretrained models [27, 39] and convert them to the desired target format using official model converters [20, 26].

*Selection of Devices and Web Browsers.* Our measurement mainly involves 50 different PC devices, featuring a diverse array of CPU models, including both Intel CPUs and Apple Silicon. Within the Intel CPU category, we encompass core-i5 and core-i7 processors spanning from Skylake to Rocket

Lake architectures [34, 35]. In the Apple Silicon category, we consider M1 and M2 CPU models. Moreover, these devices are equipped with a range of GPU models, covering both integrated and discrete graphics solutions. The discrete GPU options encompass Nvidia models such as GTX 980, 1060, and 2060, while integrated GPUs are embedded within the CPU and include variants such as Intel HD 530, 630, and 645, among others. For the Web browser, we employ Google Chrome, which stands as the most widely used browser, commanding a global market share of more than 63.5% as of September 2023 [12]. To explore the difference between different browsers, we consider Firefox to be another Web browser platform. According to browser market share data [12], Firefox holds the fourth largest share. We excluded Safari because it is only available on Apple devices, and we excluded Edge because it is also a Chromium-based browser, the same as Chrome. To explore the performance of in-browser inference on mobile devices, we involve 20 mobile devices. The mobile devices also feature a wide spectrum of **System on Chip (SoC)** models, including both Snapdragon SoC models and Kirin SoC models. The Snapdragon SoC spans from Snapdragon 855 to Snapdragon 8 Gen 3; The Kirin SoC spans from Kirin 980 to Kirin 9000. On these mobile devices, we use Chrome as the experiment Web browser platform.

## 3.2 Measurement of Inference Performance

First, we measure the performance of in-browser inference from two aspects: *latency* and *memory footprint*.

*Experiment Steps.* In the experiment, we implemented the following procedures: (1) *Kernel-level*[3] *profiling enabling*. We modified the inference framework to enable profiling kernel-level latency and memory footprint. (2) *Web service deployment*. We deployed a Web service that supports inference using various models through TF.js or ORT.js frameworks. (3) *Model inference on devices*. Devices accessed the service to perform model inference, during which we recorded the latency of each model inference. (4) We repeated step (3) multiple times and calculated the average latency and memory footprint on each device and browser, as well as the latency variance.

*Latency.* We measure the latency in both *model-level* and *kernel-level*. In terms of latency, the total inference latency consists of three types: setup latency, warmup latency, and prediction latency. (1) *Model-level latency*. We analyze the average latency gap and latency distribution among different DL frameworks, backends, and PC devices and analyze the root cause. There are different acceleration techniques in the Wasm backend, i.e., SIMD and multithreading. The two DL frameworks that we adopt both support them for acceleration. SIMD processes multiple data within a single instruction and multithreading performs computation in parallel. To explore to what extent each acceleration technique reduces the latency, we measure the latency under different settings, i.e., whether each technique is enabled or not. (2) *Kernel-level latency*. Understanding the performance of different kernels can help developers design kernel-specific optimization to improve the inference performance. For kernel-level latency, we measure the average kernel latency proportion on different backends. We do not consider the setup stage in this part because this stage does not involve kernel execution. In addition, we carry out a breakdown analysis to investigate how much SIMD and multithreading contribute to the acceleration in the Wasm backend.

*Memory Footprint.* We next measure the memory footprint, i.e., the memory usage associated with in-browser inference. Memory is a critical metric both for the inference process [95, 110] and Web applications as a whole [98]. To gain deep insights into the memory footprint of in-browser inference, we measure the memory footprint of each kernel and analyze the memory footprint growth. We omit the setup stage from our analysis since the inference execution does not take

---

[3]A DL model can be represented as a series of interconnected kernels, such as matrix multiply and convolution. The interconnections of these kernels form a computation graph.

place during this phase. Similarly, we exclude the prediction stage from consideration because the DL framework retains allocated memory for operational efficiency, resulting in a constant memory footprint throughout this phase. As a result, we measure the memory footprint of only the warmup stage.

## 3.3 Definition of QoE Metrics Dedicated to In-Browser Inference

Existing Web QoE metrics primarily focus on evaluating the speed at which a Web page becomes visually accessible during the page loading process [28, 36, 42]. In contrast, the QoE associated with in-browser inference extends its influence across the entirety of a user's browsing experience. Thus, we propose three new QoE metrics specialized for in-browser inference (including responsiveness, smoothness, and inference accuracy) and quantify it during model inference. For QoE measurement, we do not differentiate the three stages described in Figure 1, because QoE is an average metric that spans the entire Web application usage:

$$RunsPerMinute = \frac{\#Testcase}{Time_{Benchmark}} \tag{1}$$

$$fps = \frac{\#Frames_{Rendered}}{Time_{Rendering}} \tag{2}$$

$$InfAcc = \begin{cases} \frac{\#Frames_{correctlyClassified}}{\#Frames_{total}}, & classification \\ \frac{1}{N} \sum_{i=1}^{N} \int_{0}^{1} PR_i, & object\ detection \end{cases} \tag{3}$$

where $PR_i$ is Precision-Recall curve of class i, and N is #class.

*Responsiveness.* Users prefer Web pages that respond to user requests quickly. The responsiveness reflects how quickly a Web page responds to user requests. It is quantified by the response time to user requests. We measure the responsiveness of Web pages based on Speedometer [37], a benchmark for Web application responsiveness. Given the diverse application scenarios in the Web environment, it is difficult to have metrics generalize to all models or use cases. To address this challenge, the WebKit team officially provides the Speedometer dataset that covers various use cases as a benchmark. Specifically, this benchmark encompasses multiple high-level user journeys, including editing rich text, rendering charts, and reading news sites, with each journey further comprising multiple workloads for testing. In this benchmark, "runs per minute" is suggested to reflect the responsiveness QoE. As defined in Equation (1), we use the score "runs per minute" to quantify the responsiveness. The score represents the number of benchmark tests completed per unit time. We inject the inference code into the Speedometer Web page through the "Selenium" library, a library for automated Web application development and testing, allowing the developer to execute any JavaScript programs on any Web page. We perform in-browser inference during the benchmark test intermittently to avoid the benchmark test being blocked if the model inference is performed continually. During the test, model inference accounts for 50% of the total duration.

*Smoothness.* Users prefer Web pages that render smoothly without any lag. The smoothness reflects how timely the GUI content is rendered. It is quantified by the rendering/video **Frames Per Second (fps)**. As defined in Equation (2), we measure the fps of playing video and rendering Web pages. The fps stands for the average number of rendered frames per unit time during the entire testing phase. For vision tasks, we select YouTube as the target Web page for measurement; for the text task, we select Google Docs. The two Web sites are among the most popular Web sites for online videos and documents. YouTube provides continuous video streams and we can input text continuously into Google Docs. We measure the video fps in YouTube and render fps in Google Docs. We used the first recommended playback page on the YouTube homepage. Regarding Google Docs experiments, we use different input texts on the devices at varying speeds ranging from 20 to

60 words per minute. We inject the inference code into each Web page. We run the inference for 10 minutes while using the Web page and use the averaged results as the final obtained fps.

*Inference Accuracy.* Due to insufficient computation resource, inference accuracy might degrade inevitably. For example, video frames may be skipped and the application reuses previous results. Thus, as defined in Equation (3), we use inference accuracy, such as classification accuracy and detection **Mean Average Precision (mAP)**, to quantify the QoE. We use vision tasks, i.e., image classification and object detection, as the targets. We do not consider grammar checking in this part because, to our best knowledge, there lacks a benchmark dataset that contains both text and continuous timestamps. We use two benchmark datasets, including the YouTube bounding boxes dataset [54] for image classification and the YouTube Face dataset [55] for object detection. The two datasets are widely used [61, 73, 91, 96, 102, 111] because they both contain continuous real-world video stream frames and corresponding timestamps.

*Experiment Steps.* In our experiment, we executed the following procedures: (1) *Code injection using Selenium.* We utilized the Selenium [2] library to inject model inference code into various Web pages. (2) *Model inference and QoE monitoring.* After the Web pages were fully loaded, we used Selenium to initiate the model inference functions. The inference process was performed for a period, during which we recorded responsiveness and smoothness QoE. We enable multithreading and set four threads. (3) *Continuous in-browser inference simulation.* We simulated continuous in-browser inference using the two datasets and the fps values obtained from the smoothness experiment. If a video frame could be processed using the model, the inference results of that frame were used. If a frame could not be processed due to low fps or high resource usage, the results from the previous frame were re-used. This approach is quite common in the field of video processing [83, 117]. We calculate the inference accuracy after all frames are processed.

## 4  Results of Inference Performance Measurement

### 4.1  Inference Latency Analysis

*4.1.1  Model-Level Latency.* First, we compare the average inference latency in Web browsers and native environments. We deployed the Web service in a dedicated real-world server, which only serves for our experiments and is located in a LAN with experiment devices. We downloaded the model after the page loading phase because of the considerations that (1) the download time of Web pages may influence the results of the setup stage, and (2) the PLT, which is a critical indicator of the QoE, may be compromised if we download models and the framework during the page loading phase. We illustrate the results of ResNet50 in Figure 2 and provide the complete results in Tables 1–4.

*Average CPU Latency.* We observe that there is a latency gap between inference in the Wasm backend and in native. We analyze the gap of prediction latency, warmup latency, and setup latency. In this part, both SIMD and multithreading are enabled by default and we set four threads.

As for *prediction latency*, the in-browser prediction latency of TF.js is 3.7–18.4× higher than the latency of TF (Table 2 vs. Table 1); the gap is 2.1–36.4× between ORT.js and ORT (Table 3 vs. Table 1). The average gap of both frameworks is 16.9×. For both frameworks, the performance of in-browser inference is inferior consistently compared to native inference. The gap mainly comes from three aspects. First, advanced SIMD instruction sets are unavailable in browsers, like AVX2 and AVX512 on Intel CPU [22, 23]. The SIMD instruction length in Wasm is only 128-bit [47], while the length is 256-bit in AVX2 and 512-bit in AVX512. This will introduce a latency gap of up to 4× between in-browser and native inference. Besides instruction length, some advanced instructions, such as **Fused-Multiply-Add (FMA)** [19], are not available in Wasm [33]. The absence of FMA instructions in Wasm could introduce a performance gap of up to 2×. Second, the Wasm VM
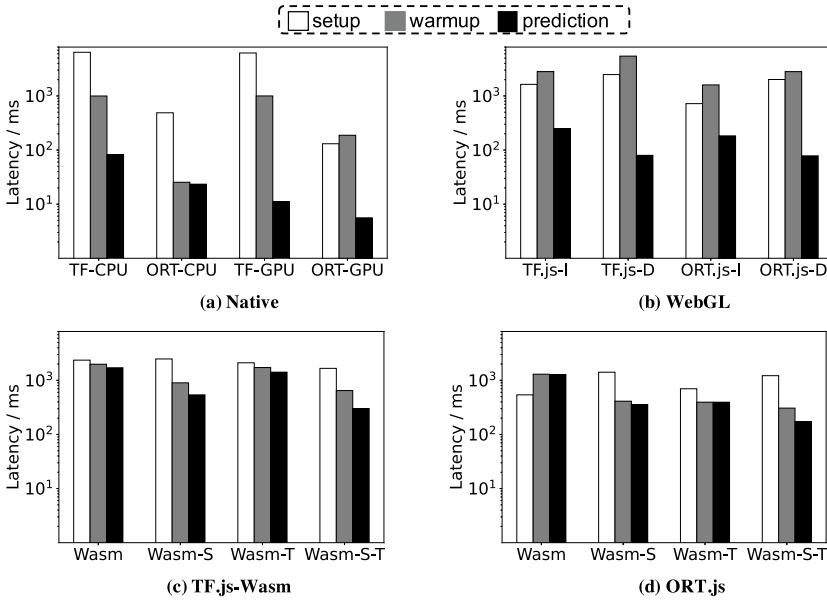
Fig. 2. ResNet50 average latency. "-I/-D" denotes integrated/discrete GPU. "-S/-T" denotes SIMD/multithreading. We found that for ResNet50, both CPU and GPU consistently offer lower native inference prediction latency. Specifically, on CPU, the average native prediction latency is 5.5× lower on average, while on GPU it is 10.5×. For both in-browser inference and native inference, using GPU achieves lower latency compared to CPU, being 3.0× and 5.8×, respectively.

Table 1. Latency (ms) of Different Models Running in Native

| | TensorFlow | | | | | | ONNX Runtime | | | | | |
| | CPU | | | GPU | | | CPU | | | GPU | | |
| | Setup | Warmup | Prediction | Setup | Warmup | Prediction | Setup | Warmup | Prediction | Setup | Warmup | Prediction |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (M1) MobileNetV2 | 1351.5 | 463.5 | 17.6 | 2069.9 | 2074.8 | 7.9 | 140.0 | 6.8 | 2.9 | 1394.2 | 1271.8 | 2.8 |
| (M2) ResNet50 | 6407.3 | 997.4 | 82.2 | 6223.4 | 1000.4 | 11.2 | 488.0 | 25.4 | 23.4 | 130.7 | 187.7 | 5.6 |
| (M3) VGG16 | 873.2 | 2308.4 | 153.6 | 3050.8 | 681.2 | 3.4 | 2057.3 | 90.4 | 101.8 | 839.8 | 126.7 | 8.2 |
| (M4) SSD-MobileNetV2 | 9359.6 | 2571.7 | 43.3 | 10031.3 | 5128.5 | 65.0 | 1245.9 | 41.7 | 29.4 | 1562.8 | 730.2 | 65.0 |
| (M5) EfficientDet | 27722.3 | 4823.6 | 482.6 | 27107.2 | 6522.6 | 228.1 | 2056.7 | 478.7 | 337.1 | 2522.4 | 9124.3 | 274.4 |
| (M6) Yolo5-Middle | 11719.9 | 1209.3 | 84.3 | 10599.7 | 1562.8 | 39.5 | 517.7 | 49.8 | 41.7 | 249.3 | 288.4 | 8.0 |
| (M7) Bert-base | 5153.0 | 856.4 | 11.7 | 4938.3 | 1196.8 | 19.1 | 614.2 | 39.6 | 31.0 | 453.8 | 18.0 | 16.0 |
| (M8) MobileBert | 29766.9 | 3526.9 | 18.5 | 30586.4 | 3269.2 | 42.5 | 970.4 | 77.4 | 53.2 | 1215.7 | 61.8 | 56.4 |
| (M9) Bert-Small | 2044.5 | 168.3 | 3.6 | 2037.6 | 109.7 | 5.0 | 212.6 | 4.6 | 2.2 | 126.7 | 6.0 | 3.8 |
| Average | 10488.7 | 1880.6 | 99.7 | 10738.3 | 2394.0 | 46.9 | 922.5 | 90.5 | 69.2 | 943.9 | 1312.8 | 48.9 |

The GPU used is the discrete one. The results show that both the TF and ORT frameworks can support the inference of these models. When performing inference on GPU, a 5.0× average lower prediction latency is achieved, but the setup and warmup latency is higher, being 1.2× and 14.1× higher compared to CPU, respectively.

introduces additional overhead, because it needs to interpret and execute the Wasm binary code, while native inference frameworks can directly run native binaries. In general, Wasm VM itself can lead up to 2.5× additional overhead [81]. Third, there are inherent threads in the Web page that introduce inevitable resource competition, such as the I/O thread and compositor thread [44].

Regarding the two in-browser inference frameworks, there is no consistent superiority of one framework over the other. For instance, we find that ORT.js exhibits lower prediction latency for ResNet50 when employing both multithreading and SIMD, while TF.js exhibits lower latency in inferring VGG16 and MobileNetV2. The difference can be primarily attributed to how the two frameworks are implemented. We delved into the implementations of both frameworks and discovered

Table 2. Latency (ms) of Supported Models in TF.js in Wasm Backend

| | Wasm | | | Wasm-thread | | | Wasm-SIMD | | | Wasm-thread-SIMD | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction |
| (M1) MobilenetV2 | 381.5 | 320.9 | 235.0 | 335.2 | 284.0 | 222.2 | 400.3 | 194.5 | 116.6 | 254.3 | 161.5 | 89.2 |
| (M2) Resnet50 | 2362.6 | 1984.3 | 1710.3 | 2109.7 | 1724.7 | 1416.1 | 2485.1 | 899.7 | 539.5 | 1665.2 | 647.3 | 300.8 |
| (M3) VGG16 | 10283.0 | 8303.8 | 6069.9 | 10265.0 | 7597.0 | 5334.9 | 10747.3 | 5482.7 | 3060.7 | 9111.0 | 2402.5 | 1021.5 |
| (M4) SSD-MobilenetV2 | 574.6 | 1386.6 | 1160.6 | 575.2 | 1174.6 | 1064.6 | 608.2 | 1024.1 | 833.4 | 474.8 | 962.4 | 795.0 |
| Average | 3400.4 | 2998.9 | 2293.9 | 3321.3 | 2695.1 | 2009.4 | 3560.2 | 1900.2 | 1137.5 | 2876.3 | 1043.4 | 551.6 |

"SIMD/thread" indicates enabled technique. We found that the Wasm backend of TF.js has insufficient support for models, supporting only four models. After enabling SIMD and multithreading, respectively, the in-browser inference latency decreased by 49.1% and 10.7%. When both were enabled, the prediction latency decreased by 64.7%, demonstrating the potential of both technologies in accelerating inference.

Table 3. Latency (ms) of Models in ORT.js Wasm Backend

| | Wasm | | | Wasm-thread | | | Wasm-SIMD | | | Wasm-thread-SIMD | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction |
| (M1) MobilenetV2 | 581.4 | 148.1 | 120.6 | 1306.6 | 216.9 | 122.4 | 1252.1 | 162.8 | 106.5 | 331.0 | 139.6 | 105.5 |
| (M2) Resnet50 | 539.3 | 1299.5 | 1275.3 | 699.1 | 395.1 | 394.7 | 1412.3 | 411.9 | 358.1 | 1219.3 | 306.9 | 172.3 |
| (M3) VGG16 | 7545.3 | 9024.7 | 8275.6 | 4437.5 | 8147.2 | 8140.1 | 2695.2 | 3164.6 | 2947.2 | 2423.2 | 1353.6 | 1334.1 |
| (M4) SSD-MobilenetV2 | 1120.5 | 287.6 | 279.4 | 1338.1 | 197.3 | 123.9 | 1171.8 | 174.5 | 126.1 | 1467.8 | 84.5 | 81.4 |
| (M5) EfficientDet | 1581.2 | 2665.4 | 2648.6 | 1786.4 | 1251.6 | 1282.5 | 1606.1 | 1245.8 | 1228.8 | 1889.6 | 926.0 | 852.4 |
| (M6) Yolo5-Middle | 427.6 | 1105.2 | 1092.7 | 450.0 | 365.9 | 362.9 | 440.8 | 406.2 | 386.4 | 457.6 | 145.8 | 145.2 |
| (M7) Bert-base | 744.6 | 915.2 | 910.2 | 702.4 | 285.8 | 279.6 | 681.8 | 246.2 | 247.2 | 574.8 | 92.8 | 81.6 |
| (M8) MobileBert | 873.5 | 855.1 | 849.8 | 935.8 | 310.1 | 291.1 | 865.0 | 239.3 | 236.2 | 945.0 | 120.2 | 113.6 |
| (M9) Bert-Small | 155.2 | 21.2 | 21.8 | 161.3 | 9.2 | 7.8 | 158.1 | 7.1 | 6.7 | 166.8 | 4.6 | 4.7 |
| Average | 1507.6 | 1813.6 | 1719.3 | 1313.0 | 1242.1 | 1222.8 | 1138.9 | 673.2 | 627.0 | 1052.8 | 352.7 | 321.2 |

"-SIMD/thread" indicates enabled technique. We found that the Wasm backend of ORT.js supports all nine models. After SIMD and multithreading technologies were enabled, the prediction latency decreased by 50.7% and 31.2%, respectively. When both were enabled, the prediction latency decreased by 63.4%.

Table 4. Latency (ms) of Models in WebGL Backend

| | ORT.js-WebGL-I | | | ORT.js-WebGL-D | | | TF.js-WebGL-I | | | TF.js-WebGL-D | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction |
| (M1) MobilenetV2 | 210.7 | 2094.4 | 33.1 | 306.6 | 2866.8 | 31.4 | 209.5 | 2900.7 | 28.7 | 530.9 | 5751.0 | 20.1 |
| (M2) Resnet50 | 720.6 | 1599.6 | 183.0 | 2011.8 | 2810.6 | 78.2 | 1635.7 | 2814.2 | 248.3 | 2482.7 | 5443.6 | 79.8 |
| (M3) VGG16 | 733.5 | 1793.5 | 545.6 | 2086.3 | 2538.4 | 146.4 | 9109.0 | 3414.5 | 1748.7 | 10090.0 | 4719.0 | 913.9 |
| (M4) SSD-MobilenetV2 | / | / | / | / | / | / | 503.1 | 7047.6 | 1039.0 | 1125.2 | 9097.9 | 616.6 |
| (M5) EfficientDet-D1 | / | / | / | / | / | / | 734.8 | 9235.6 | 2736.1 | 2542.3 | 11127.3 | 1342.7 |
| (M6) Yolo5-Middle | / | / | / | / | / | / | 1739.7 | 5796.1 | 99.4 | 2054.3 | 10873.6 | 82.9 |
| (M7) Bert-base | / | / | / | / | / | / | 1902.7 | 3126.1 | 682.0 | 3412.0 | 3128.2 | 133.4 |
| (M8) MobileBert | / | / | / | / | / | / | 2837.3 | 3169.5 | 943.1 | 3538.0 | 2196.4 | 400.5 |
| (M9) Bert-Small | / | / | / | / | / | / | 664.0 | 2089.0 | 41.1 | 839.3 | 2273.4 | 58.3 |
| Average | 554.9 | 1829.2 | 253.9 | 1468.2 | 2738.6 | 85.3 | 2148.4 | 4399.3 | 840.7 | 2957.2 | 6067.8 | 405.4 |

"-I/-D" denotes integrated/discrete GPU. "/" indicates the model is not supported by the framework and backend. We found that the two frameworks differ in their support for models on the WebGL backend. Specifically, the WebGL backend of TF.js provides better support for models, supporting six more models than ORT.js. The prediction latency using a discrete GPU is lower than that using an integrated GPU, being 38.9% and 45.2% lower for TF.js and ORT.js, respectively. However, the warmup latency on GPU is more significant, being 43.5% higher than the prediction latency for both frameworks and both types of GPU.

that both frameworks use XNNPACK for the underlying kernel computation, where XNNPACK is a library that provides numerous kernel implementations. However, there are differences in how a computation graph is constructed at a higher level, as well as the kernels themselves. For TF.js, graph parsing and construction, as well as kernel scheduling, are implemented on the JavaScript side, with subsequent calls to XNNPACK [5] kernels for execution. In contrast, ORT.js implements these within Wasm. Furthermore, TF.js utilizes models directly converted from TF SavedModels [3] for loading and graph construction, whereas ORT.js requires a transformation of the SavedModel to adapt it to the specifications of ORT and ORT.js. These implementation differences lead to varied

performance across different models between the two frameworks. Besides, TF.js shows wider model support compared with ORT.js.

As for *warmup latency*, the in-browser warmup latency of TF.js is 0.4–1.0× of the latency of TF (Table 2 vs. Table 1), while the gap is 1.0–20.5× for ORT.js and ORT (Table 3 vs. Table 1). TF suffers from higher warmup latency across all models compared to TF.js. This is because loading all necessary components, such as the compute graph engine, of TF for model inference requires more than 980 MB of RAM size, while the library size of TF.js is only 3 MB, indicating that the RAM size will not exceed 3 MB. During the warmup stage, the primary tasks include allocating memory for inference and initializing execution. Since memory allocation is not influenced by underlying system software and hardware instruction characteristics, the warmup latency gap between the browser and the native is not as significant as the gap in prediction latency. When comparing the warmup latency with the prediction latency, we find that warmup latency is 1.2–2.3× of the average prediction latency for TF.js (Table 2), while the gap is 1.0–1.8× for ORT.js (Table 3). Warmup additional overhead primarily arises from memory allocation in Wasm. After the warmup stage, the frameworks maintain the allocated memory for efficiency. The cold start issue is less severe in ORT.js compared to TF.js. This is primarily because ORT.js optimizes the computation graph, reducing the frequency of memory allocations. Additionally, ORT.js leverages an Arena memory pool for further optimization [30, 31].

As for *setup latency*, the in-browser inference setup latency of TF.js is 0.1–10.4× of the latency of TF (Table 1 vs. Table 2), while it is 0.9–2.4× for ORT.js and ORT (Table 1 vs. Table 3). It is worth noting that the setup stage of native inference only involves loading the model and the framework. In-browser inference may outperform native inference w.r.t. the setup latency. One overhead of setup arises from loading the framework into memory. The native inference framework itself has a larger size. For example, simply importing TF in Python (i.e., executing Python command "import tensorflow," which is performed in the setup stage) requires over 160 MB of RAM size, while the required files are only about 3 MB for TF.js. On the other hand, native inference may also outperform in-browser inference because in-browser inference needs to download model files to the browser. The download latency depends on the network and model size. For example, in TF.js, downloading VGG16 requires downloading 528 MB of data, which introduces high network transfer latency. Among the selected models, the model file size varies up to 31.1×. The significant disparity in data transfer latency results in a wide range of setup latency.

To further explore the impact of network conditions during the setup stage, we deployed a monitor on the server to observe the network latency for each device as it downloaded the model and framework. To avoid competition for network bandwidth among devices, we ensured that only one device was downloading the model at any given time. We observed relatively stable changes in network bandwidth, with the bandwidth for transferring files between different devices ranging from 51.2 to 76.4 MB/s. Regarding the latency in file transmission, the latency varied from 212 ms to 9,717 ms. We compared this data transfer time with the setup stage duration and found that on average, data transmission latency constituted 88.5% of the setup latency, indicating that data transmission dominates this stage and is the main bottleneck.

In summary, by analyzing data from Table 1 regarding CPU performance and Wasm-thread-SIMD data from Tables 2 and 3, several key findings emerge. Firstly, for prediction latency, a significant gap persists between in-browser and native inference, primarily due to performance differences between native and browser environments. The lack of advanced SIMD instructions, such as FMA in the Wasm environment, results in lower acceleration compared to native inference, with additional impacts from resource competition among inherent browser threads. Secondly, regarding the warmup latency, TF's substantial library size leads to exceptionally higher latency than TF.js. However, warmup latency is generally higher in all settings than prediction latency, reaching up to

2.3× higher, primarily due to the need to load and initialize programs and allocate memory during the warmup stage. Lastly, setup latency is primarily influenced by network conditions. In-browser inference involves downloading models from servers, with model data transfer latency constituting a significant portion of setup latency, reaching 88.5% on average. Despite this, current frameworks support download and load pipelining, which allows parallel operations, enhancing the efficiency of the setup stage.

*Breakdown Analysis for Wasm Backend.* We explore the influence of multithreading and SIMD on latency. These two techniques are to accelerate computation, significantly impacting both the warmup and prediction stages. We set four threads when enabling multithreading. The results of ResNet50 are illustrated in Figure 2 and the full results are presented in Tables 2 and 3. The results show that SIMD reduces 49.1% and 50.7% average prediction latency for TF.js and ORT.js, respectively, while multithreading reduces 10.7% and 31.2% average prediction latency for TF.js and ORT.js, respectively. When enabling both techniques, the average prediction latency reduces by 64.7% and 63.4% for TF.js and ORT.js, respectively. For both frameworks, SIMD provides a greater speedup compared to multithreading. This is mainly because multithreading synchronization is implemented through message passing in Wasm, introducing additional overhead for thread management and synchronization [21, 43]. Besides, inherent resource competition from other threads in the browser further slows down the inference, such as the I/O thread and compositor thread in Chromium architecture. For example, for a CPU with four cores, if we have allocated four threads for inference, the remaining threads will compete with the inference threads for resources, thereby impacting the efficiency of the inference process. Both multithreading and SIMD can provide up to 4× acceleration (since 128-bit instruction length is 4× the length of int32/float32 data; single thread vs. four threads). However, the two technologies only achieve a maximum acceleration of 2.8× for inference. This is because the model contains many memory-intensive kernels (37.7% on average across all models), and both frameworks provide limited optimization for these kernels.

Regarding the warmup latency, it was found that SIMD and multithreading, respectively, reduce latency by 38.5% and 12.1% on TF.js, and by 54.5% and 41.5% on ORT.js. These reductions are lower than those observed during the prediction stage, primarily because the warmup stage also involves memory allocation operations, which are less affected by these technologies. However, since computation still dominates during the warmup stage, both technologies are still able to facilitate acceleration.

In summary, an acute analysis of data from Tables 2 and 3 reveals that for in-browser inference, SIMD can provide up to 54% lower latency on average compared to multithreading. This improvement is mainly due to the efficiency of synchronization between threads and the impact of resource competition. However, the presence of memory-intensive operations limits the maximum acceleration achievable by both technologies to only 2.8×, as both SIMD and multithreading offer limited speed enhancements for memory-intensive operations.

*Average GPU Latency.* We analyze the latency in WebGL and compare it with the latency in native and Wasm in the same three latency categories. Note that when comparing with native inference, we only consider discrete GPUs because the native DL framework does not support the integrated one.

As for *prediction latency*, TF.js is 2.5–268.7× slower than TF, and the gap between ORT.js and ORT is 11.2–17.9× (Table 1 vs. Table 4). The average gap for both frameworks is 30.6×. The gap mainly arises from inefficient libraries and GPU abstraction. Native inference frameworks can utilize high-performance GPU libraries, such as CUDA [16], to fully exploit the GPU's parallelism capacity. CUDA's low-level access to GPU allows developers to interact directly with the hardware, enabling fine-grained control and optimized program execution. However, in-browser inference frameworks rely on WebGL. WebGL is primarily designed for rendering tasks and requires using

graphics-related mechanisms like explicit textures and frame buffers. Developers cannot access GPU directly through WebGL, making it inefficient for parallel computing tasks.

When comparing the latency in the WebGL backend with the Wasm backend, we also find that WebGL exhibits lower latency than Wasm. For TF.js, inference in WebGL on integrated and discrete GPUs results in 1.4× and 2.7× average prediction latency reduction, respectively, compared to Wasm. For ORT.js, the reduction is 2.2× and 4.9×, respectively. The speedup contributes to the GPU capacity, despite the limitations of WebGL in fully utilizing the hardware features.

As for *warmup latency*, we observed that the warmup latency for in-browser inference is significantly high. Specifically, when compared to native inference, TF.js is on average 5.8× slower than TF, and the average gap between ORT.js and ORT is 186.7× (Table 1 vs. Table 4). Additionally, when comparing warmup latency to prediction latency, we found that the warmup latency of TF.js on discrete GPUs is 64.6× longer than its prediction latency, while on integrated GPUs, the gap is 26.8×. For ORT.js, these gaps are 48.1× and 25.1×, respectively (as detailed in Table 4). In-browser inference exhibits severe warmup latency in WebGL because the shader is compiled in this stage [52]. Taking Chrome on the Windows platform as an example, when executing WebGL, Chrome uses ANGLE to compile WebGL shader code into an intermediate representation, such as Direct3D, which is then compiled by the GPU hardware into executable binaries [1]. This compilation process is completed during the first execution of WebGL, and the compiled results can be re-used in subsequent executions. The compilation process is time-consuming. Therefore, in-browser inference using the WebGL backend experiences high warmup latency. In addition, memory allocation also introduces overhead in this stage. Integrated GPU outperforms discrete GPU because the inference process needs to invoke the GPU and synchronize the process state [50, 51] during the initializing execution of WebGL, which is slow on discrete GPU. We also find that ORT.js outperforms TF.js by 1.7× on integrated GPU and 1.9× on discrete GPU, because ORT.js optimizes the model via kernel fusion during model conversion, resulting in shorter compilation time.

As for *setup latency*, TF.js is 0.1–3.3× of TF, and the gap is 0.2–15.4× for ORT.js and ORT (Table 1 vs. Table 4). The reason is similar to that of Wasm. We also find that for in-browser inference, integrated GPU outperforms discrete GPU consistently. Specifically, the gap is 1.1–3.5× for TF.js and 1.5–2.8× for ORT.js (Table 4). This is mainly because of high invoking overhead on discrete GPU [50, 51]. Unlike the invocation during the warmup stage, the invocation in the setup stage primarily involves loading the model parameters and the inference framework onto the GPU.

In summary, based on the analysis of data from the GPU part of Tables 1 and 4, we find that using WebGL for inference is a faster solution compared to Wasm. Compared with Wasm, WebGL achieves up to 2.7× faster prediction than Wasm. However, compared to native inference using CUDA, WebGL exhibits a significantly larger performance gap, up to 268.7×, due to its inherent inefficiencies. Different from Wasm, using WebGL involves in-browser compilation of WebGL shaders during the warmup stage, which results in a severe warmup latency, up to 64.6× higher than the prediction latency. Regarding the setup stage, invoking a discrete GPU incurs higher latency.

*Latency Variance.* Figures 3 and 4 show the prediction latency and warmup latency variance, respectively. The variance is the latency gap between the highest and the lowest average latency among devices. We exclude setup latency variance due to the impact of different network conditions.

The *prediction latency* varies differently among devices. Take ResNet50 in the Wasm backend of TF.js as an example, the latency quartiles are 257.6 ms, 358.4 ms, and 680.9 ms. The prediction latency variance of all models in the Wasm backend reaches up to 28.4×. Such variance mainly derives from hardware capacities. We observe that the benchmark score of the lowest-end CPU involved, Core i5 Skylake, is 4.1× lower than the highest-end CPU, Core i7 Rocket Lake (6,030 vs. 24,673) for one core [15]. Moreover, the same kernel can also exhibit different performances

**(a) ORT.js-Wasm pred. latency/ms**


**(b) ORT.js-WebGL pred. latency/ms**


**(c) TF.js-Wasm pred. latency/ms**


**(d) TF.js-WebGL pred. latency/ms**
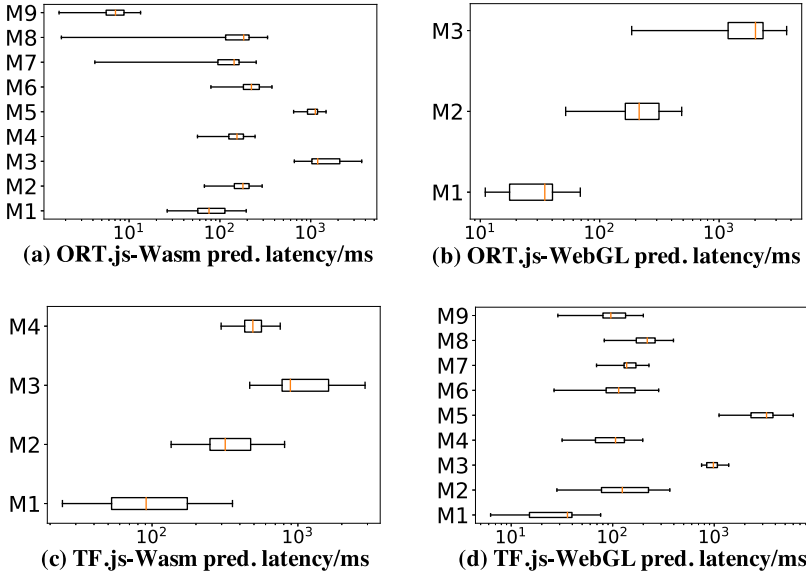
Fig. 3. Prediction latency distribution. "M*" is model ID and is the same with Section 3.1. We found that the variance in prediction latency reached 28.4× on the Wasm backend and 19.4× on the WebGL backend for all models and both frameworks, primarily due to differences in device hardware performance.


**(a) ORT.js-Wasm warmup latency/ms**


**(b) ORT.js-WebGL warmup latency/ms**


**(c) TF.js-Wasm warmup latency/ms**


**(d) TF.js-WebGL warmup latency/ms**
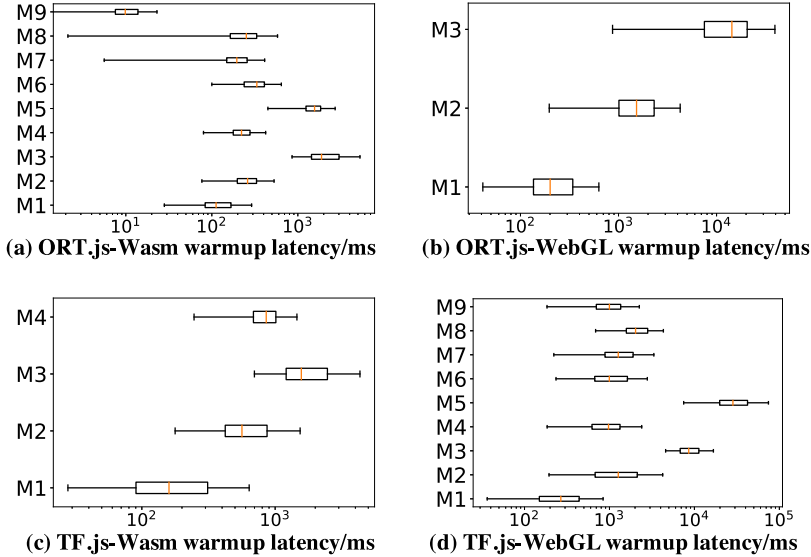
Fig. 4. Warmup latency distribution. "M*" is model ID and is the same with Section 3.1. We found that the variance in warmup latency reached 25.3× on the Wasm backend and 14.4× on the WebGL backend for all models and both frameworks.

on different hardware [65, 112]. In the WebGL backend in TF.js, the prediction latency quartiles of ResNet50 are 77.5 ms, 124.3 ms, and 225.3 ms. The variance of all models reaches up to 19.4×. Such variance also arises from the large device capacity gap. We observe that the score of the highest-end GPU in our dataset, Nvidia GTX 2060, is 207.4× better than the lowest-end Intel HD

530 according to the GPU benchmark [38]. Since WebGL has limited capacity to harness discrete GPUs, the variance is less pronounced compared to the differences in GPU capacity.

Regarding the performance differences between different models on integrated GPUs, we compared the Apple Silicon GPUs with GPUs integrated with Intel chips. We observed that Apple Silicon GPUs generally outperform those in Intel chips. Specifically, for TF.js, prediction latency on Apple Silicon GPUs is on average 7.1× lower than that on Intel chips; for ORT.js, the latency on Apple Silicon GPUs is on average 6.2× lower than that on Intel chips. This enhanced performance is primarily due to the optimizations in the Apple GPU for parallel computing, which makes it more adept at handling model inference tasks. In summary, for in-browser inference, the standards of Wasm and WebGL are consistent across all devices, indicating that any device capable of running a Web browser can support in-browser inference. The differences in performance are indeed related to the capabilities of the different devices or hardware types, such as the capacity gap between integrated and discrete GPUs.

Regarding the performance differences in in-browser inference across different CPU architectures, we compared the inference capabilities between ARM CPUs and x86 CPUs. For ARM architecture, we selected Apple Silicon, with our dataset including both the M1 and M2 chips, while for x86 architecture, we used Intel CPUs. We found that the latency on ARM CPUs is on average lower than x86 CPUs. Specifically, for TF.js, the latency on ARM CPUs was 2.2× slower than x86 CPUs, and for ORT.js, the gap is 2.9×. This disparity is significantly smaller than the previously analyzed factor of 28.6×. The primary reason for this smaller gap is the considerable variation in the performance of the Intel chips used in our dataset, which ranged from Core-i5 Skylake models to Core-i7 Rocket Lake models. Lower-end Intel chips performed worse than Apple chips, whereas higher-end Intel models surpassed the performance of Apple chips. In summary, there is a greater variance in inference performance across different Intel CPUs, and the average performance gap between Intel and Apple CPUs is relatively small. The highest-end Intel CPUs demonstrated the best inference performance.

As for *warmup latency*, the variance of all models reaches up to 25.3× in the Wasm backend and 14.4× in the WebGL backend. The variance is not as significant as the prediction latency. The reason is that the warmup latency consists of memory allocation overhead. Compared to computation, the impact of hardware capacity on memory allocation is negligible. Consequently, when memory allocation is involved, the variance of warmup latency decreases compared to the variance in prediction latency. However, warmup latency remains predominantly driven by computation, i.e., initializing the first model inference. Therefore, due to significant performance differences between hardware capacities, the variance in warmup latency can still reach up to 25.3×.

In summary, after analyzing the results of Figures 3 and 4 the variance in prediction latency and warmup latency across devices is quite significant, primarily attributed to variations in the hardware capacity of user devices, as both types of latency are predominated by computation. Compared to prediction latency, the variance in warmup latency across devices is relatively smaller. This is mainly because warmup latency also encompasses memory allocation, where the performance variance across devices is minimal, resulting in this observation.

*Model-Level Latency Difference across Different Models and Tasks.* We explore the latency difference across different models within the same task. Taking prediction latency as an example, we found that MobileNetV2 exhibits the lowest latency in image classification tasks in all settings, while Bert-Small shows the lowest latency in grammar checking tasks in all settings. This is primarily because these two models are the lightest within their respective task model sets. Specifically, MobileNetV2 has a model file size of only 14 MB, and Bert-Small is only 17 MB. However, for object detection tasks, SSD-MobileNetV2 performs best in the Wasm backend of ORT.js, while Yolo5-Middle shows the best performance in the WebGL backend of TF.js. This is mainly due to the differences in the

Table 5. Latency (ms) of Supported Models in TF.js in Wasm Backend in Firefox Browser

| | Wasm | | | Wasm-thread | | | Wasm-SIMD | | | Wasm-thread-SIMD | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction |
| (M1) MobilenetV2 | 408.6 | 345.9 | 250.8 | 351.4 | 313.1 | 239.5 | 402.1 | 205.1 | 123.5 | 260.8 | 172.1 | 93.9 |
| (M2) Resnet50 | 2529.2 | 2102.9 | 1905.0 | 2261.8 | 1936.4 | 1574.4 | 2546.9 | 993.0 | 592.4 | 1766.0 | 743.0 | 341.0 |
| (M3) VGG16 | 10110.7 | 8803.6 | 6701.2 | 10417.8 | 8606.1 | 5803.8 | 10856.9 | 6090.2 | 3294.7 | 9182.0 | 2641.1 | 1132.5 |
| (M4) SSD-MobilenetV2 | 612.1 | 1532.9 | 1249.2 | 571.3 | 1252.0 | 1195.4 | 644.5 | 1148.6 | 922.7 | 498.3 | 1073.5 | 835.6 |
| Average | 3415.2 | 3196.3 | 2526.6 | 3400.6 | 3026.9 | 2203.3 | 3612.6 | 2109.2 | 1233.3 | 2926.8 | 1157.4 | 600.8 |

"-SIMD/thread" indicates enabled technique. The results show that the Wasm backend of TF.js can support only four models, indicating limited compatibility. Enabling SIMD and multithreading technologies resulted in decreases in in-browser inference prediction latency by 49.2% and 9.9% on average, respectively. When both techniques were applied, they achieved a further reduction in prediction latency by 65.2%, confirming their effectiveness in optimizing in-browser inference performance.

Table 6. Latency (ms) of Models in ORT.js Wasm Backend in Firefox Browser

| | Wasm | | | Wasm-thread | | | Wasm-SIMD | | | Wasm-thread-SIMD | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction |
| (M1) MobilenetV2 | 589.6 | 159.7 | 132.8 | 1389.0 | 233.6 | 134.9 | 1276.4 | 175.2 | 115.5 | 355.1 | 151.8 | 112.6 |
| (M2) Resnet50 | 558.6 | 1509.5 | 1363.6 | 711.2 | 434.8 | 455.7 | 1423.6 | 458.4 | 398.5 | 1271.0 | 360.0 | 191.1 |
| (M3) VGG16 | 7612.3 | 10248.4 | 9627.2 | 4790.3 | 8853.1 | 8932.5 | 2797.2 | 3331.8 | 3236.4 | 2444.9 | 1429.4 | 1444.8 |
| (M4) SSD-MobilenetV2 | 1165.4 | 304.9 | 295.5 | 1425.3 | 229.8 | 136.4 | 1173.5 | 186.8 | 140.9 | 1533.6 | 93.0 | 95.3 |
| (M5) EfficientDet | 1640.8 | 2893.8 | 2821.9 | 1898.6 | 1384.2 | 1434.3 | 1710.9 | 1427.4 | 1330.1 | 1961.7 | 1029.6 | 920.6 |
| (M6) Yolo5-Middle | 428.9 | 1186.3 | 1280.6 | 477.4 | 406.5 | 412.5 | 421.4 | 460.1 | 450.0 | 467.9 | 160.9 | 167.1 |
| (M7) Bert-base | 804.1 | 1017.8 | 1020.3 | 744.2 | 310.5 | 296.3 | 716.4 | 287.1 | 290.3 | 576.7 | 100.6 | 90.1 |
| (M8) MobileBert | 875.0 | 928.8 | 930.0 | 946.0 | 351.2 | 334.2 | 868.7 | 259.6 | 257.4 | 959.0 | 135.6 | 124.0 |
| (M9) Bert-Small | 155.9 | 22.6 | 24.8 | 172.1 | 10.2 | 8.7 | 167.5 | 7.7 | 7.8 | 171.0 | 4.9 | 5.3 |
| Average | 1536.7 | 2030.2 | 1944.1 | 1394.9 | 1357.1 | 1349.5 | 1172.8 | 732.7 | 691.9 | 1082.3 | 385.1 | 350.1 |

"-SIMD/thread" indicates enabled technique. We found that ORT.js's Wasm backend supports all the nine models tested. Upon enabling SIMD and multithreading, we recorded reductions in prediction latency of 59.2% and 49.2% on average, respectively. A combination of both acceleration techniques resulted in an additional latency decrease of 73.8% compared to vanilla Wasm.

implementations of the two backends across the two frameworks. We also observed significant differences in prediction latency among different models. Taking image classification as an example, in our experiments, the file sizes for the MobileNetV2 and ResNet50 models were 14 MB and 98 MB, respectively. Additionally, the primary kernels included in these models differ: MobileNetV2 mainly incorporates depthwise separable convolutions and pointwise convolutions, whereas ResNet50 predominantly uses $3 \times 3$ convolutions. These structural differences between the two models lead to variations in setup latency, warmup latency, and prediction latency. As for models used in different tasks, take for example the image classification model MobileNetV2 and the object detection model SSD-MobileNetV2. Although SSD-MobileNetV2 uses MobileNetV2 as its feature extractor, it omits the last four bottleneck layers of MobileNetV2 and incorporates additional detection layers. Such structural differences between the two models affect their inference latency. In summary, the differences in inference latency between models are primarily due to variations in their model structure.

*Results on Firefox Browser.* We further investigated the impact of different browsers. For each application category, we selected a model and presented the results in Tables 5–7.

*Starting with the CPU Results.* The first is prediction latency. By comparing Tables 5 and 6 with Table 1, we still observed that a gap still exists between in-browser inference and native inference, averaging 8.8×. This is primarily due to the differences between the browser and native environments, including the absence of advanced instructions and inevitable resource competition. Additionally, we compared in-browser inference between Firefox and Chrome (Tables 5 and 6 vs. Tables 2 and 3). We found that for the same framework, the Chrome browser outperforms the Firefox browser consistently. Specifically, for TF.js, the average prediction latency on Firefox is 1.08× that of Chrome; for ORT.js, it is 1.11×. The second is warmup latency. Compared with

Table 7.   Latency (ms) of Models in WebGL Backend in Firefox Browsers

| | ORT.js-WebGL-I | | | ORT.js-WebGL-D | | | TF.js-WebGL-I | | | TF.js-WebGL-D | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction |
| (M1) MobilenetV2 | 225.4 | 2225.2 | 34.1 | 313.6 | 3299.9 | 32.9 | 222.6 | 3312.2 | 30.9 | 579.5 | 6072.4 | 21.3 |
| (M2) Resnet50 | 769.9 | 1825.3 | 207.3 | 2145.9 | 3122.1 | 80.2 | 1668.6 | 2927.9 | 265.9 | 2679.9 | 5794.3 | 85.9 |
| (M3) VGG16 | 737.5 | 1884.5 | 617.4 | 2257.1 | 2825.5 | 166.4 | 9340.9 | 3834.8 | 1999.3 | 10618.5 | 5431.4 | 1028.8 |
| (M4) SSD-MobilenetV2 | / | / | / | / | / | / | 511.1 | 7510.0 | 1183.9 | 1196.2 | 10075.0 | 664.2 |
| (M5) EfficientDet-D1 | / | / | / | / | / | / | 772.1 | 9914.6 | 2845.2 | 2654.2 | 12557.6 | 1521.1 |
| (M6) Yolo5-Middle | / | / | / | / | / | / | 1895.3 | 6487.3 | 101.8 | 2173.0 | 11244.7 | 94.5 |
| (M7) Bert-base | / | / | / | / | / | / | 2014.7 | 3251.5 | 697.4 | 3574.9 | 3598.9 | 141.2 |
| (M8) MobileBert | / | / | / | / | / | / | 3100.0 | 3397.4 | 1079.5 | 3748.9 | 2504.1 | 426.2 |
| (M9) Bert-Small | / | / | / | / | / | / | 709.7 | 2250.9 | 45.8 | 848.4 | 2578.4 | 61.1 |
| Average | 577.6 | 1978.3 | 286.3 | 1572.2 | 3082.5 | 93.2 | 2248.3 | 4765.2 | 916.6 | 3119.3 | 6650.8 | 449.4 |

"/" indicates the same as Table 4. The results demonstrate that the WebGL backend of TF.js can accommodate all nine models. Furthermore, the warmup latency for performing inference on GPU surpasses the prediction latency, with average increases of 45.8× for TF.js and 38.9× for ORT.js.

native inference latencies, the results are similar to those on Chrome, i.e., in-browser inference warmup latency is on average 0.37–1.14× of native inference for TF.js; the gap for ORT.js and ORT is 1.07–22.32×. The difference between browsers remains low, with Firefox's results varying from Chrome's by no more than 10.3% for both frameworks. The main difference between the two browsers with respect to warmup and prediction latency arises from the implementation of the browsers because the standard of Wasm remains the same between the two browsers. The third is setup latency. Compared to native inference, the setup latency for TF.js is 0.05–10.5× that of TF, and for ORT.js it is 0.84–2.6× that of ORT. Since in-browser inference setup latency is mainly influenced by library size and network conditions, there are significant differences between models as well as between native and browser environments. Among different browsers, the setup latency on Firefox is 1.03× that of Chrome, which is slightly lower than prediction and warmup latencies. This is because the setup stage is predominantly by network conditions as previously analyzed. In summary, the difference between the two browsers is minimal and can be considered negligible compared to the native vs. browser differences.

Furthermore, we analyzed the impact of different acceleration technologies, specifically multi-threading and SIMD. As with previous experiments, we set up four threads. We found that SIMD can reduce the prediction latency for TF.js and ORT.js by 49.2% and 59.2%, respectively, while multithreading can reduce it by 9.9% and 49.2% for TF.js and ORT.js, respectively. When both technologies were enabled, the prediction latency was reduced by 65.2% and 73.8% for TF.js and ORT.js, respectively. Similar to Chrome, the reason multithreading is less effective than SIMD is due to unavoidable resource competition and the overhead of thread synchronization, which is determined by the Wasm itself and is independent of the browser architecture.

Next, we analyzed the average latency on the GPU. Firstly, the setup latency for in-browser inference is 0.1–12.2× that of native inference for TF.js, and the gap is 0.2–16.4× for ORT.js and ORT. Compared to Chrome, Firefox's setup latency on a discrete GPU is 1.06× on average higher, and on an integrated GPU, the average gap is 1.05×, which is negligible compared with the gap between native inference. Secondly, as for warmup latency, compared to native inference, the average in-browser inference warmup latency is 6.1× that of native inference on average for TF.js and the gap is 13.8× for ORT.js, primarily due to time-consuming in-browser WebGL shader compilation; compared to Chrome, Firefox's data average 1.11× on discrete GPU and 1.08× on integrated GPU. Finally, compared to native inference, the prediction latency for in-browser inference averages 40.2× that of native inference for TF.js, and the average gap is 15.5× for ORT.js; compared to Chrome, Firefox's data average 1.09× higher for both frameworks. We found that similar to the results on the CPU, Firefox's performance is somewhat inferior to Chrome's. This is primarily due

Table 8. Latency (ms) of Different Models Running in Native on Mobile Devices

| | TFLite | | | | | | mORT | | | | | |
| | | CPU | | | GPU | | | CPU | | | GPU | |
| | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (M1) MobilenetV2 | 22.0 | 21.3 | 7.8 | 1123.8 | 17.6 | 9.9 | 186.7 | 37.7 | 24.8 | 54.2 | 29.1 | 13.7 |
| (M2) Resnet50 | 134.1 | 130.9 | 92.9 | 1478.4 | 68.4 | 54.7 | 265.4 | 131.0 | 100.3 | 133.0 | 84.0 | 66.7 |
| (M3) VGG16 | 668.6 | 425.8 | 403.6 | 3484.5 | 140.9 | 121.7 | 1386.4 | 366.1 | 357.4 | 935.4 | 228.9 | 216.0 |
| (M4) SSD-MobilenetV2 | 658.1 | 138.0 | 130.4 | / | / | / | 1420.7 | 125.2 | 95.4 | 691.1 | 65.2 | 47.7 |
| (M5) EfficientDet | 1103.5 | 9983.9 | 1339.1 | / | / | / | 1570.2 | 446.1 | 407.2 | 1570.2 | 446.1 | 407.2 |
| (M6) Yolo5-Middle | 571.5 | 295.8 | 259.4 | 2424.0 | 74.2 | 64.7 | 765.1 | 153.6 | 125.3 | 371.8 | 98.7 | 72.8 |
| (M7) Bert-base | / | / | / | / | / | / | 1167.2 | 127.1 | 72.1 | 572.3 | 57.1 | 34.9 |
| (M8) MobileBert | / | / | / | / | / | / | 1629.2 | 104.0 | 82.7 | 976.0 | 69.4 | 47.5 |
| (M9) Bert-Small | 17.8 | 8.7 | 3.6 | / | / | / | 160.0 | 15.1 | 9.2 | 160.0 | 9.3 | 5.9 |
| Average | 453.7 | 1572.1 | 319.5 | 2127.7 | 75.3 | 62.8 | 950.1 | 167.3 | 141.6 | 607.1 | 120.9 | 101.4 |

"/" indicates that the model is not supported. We observed that TFLite offers less comprehensive model support compared to mORT on both CPU and GPU. Both frameworks facilitate GPU acceleration, resulting in GPU inference achieving lower average prediction latencies compared to CPU by 2.5× for TFLite and 1.7× for mORT for the models that are supported by both frameworks in each backend. For TFLite, the prediction latency averages 319.5 ms on the CPU and 62.8 ms on the GPU; for mORT, the prediction latency averages 141.6 ms on the CPU and 101.4 ms on the GPU.

Table 9. Latency (ms) of Supported Models in TF.js in Wasm Backend on Mobile Devices

| | Wasm | | | Wasm-thread | | | Wasm-SIMD | | | Wasm-thread-SIMD | | |
| | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (M1) MobilenetV2 | 4163.9 | 618.7 | 532.4 | 4238.9 | 652.7 | 484.9 | 4431.0 | 667.4 | 428.5 | 3691.0 | 463.1 | 324.7 |
| (M2) Resnet50 | 23855.5 | 3867.0 | 3370.6 | 26953.4 | 3205.1 | 2838.8 | 30543.8 | 1745.2 | 1187.1 | 18561.5 | 926.2 | 846.3 |
| (M3) VGG16 | 125750.1 | 15957.2 | 13540.6 | 131675.1 | 13852.1 | 11047.3 | 133282.5 | 10018.9 | 6435.4 | 101524.8 | 4681.6 | 3236.3 |
| (M4) SSD-MobilenetV2 | 7749.1 | 3025.4 | 2977.1 | 7174.2 | 3161.8 | 2669.8 | 7557.0 | 2910.0 | 2736.0 | 7025.8 | 2843.5 | 2191.4 |
| Average | 40379.7 | 5867.1 | 5105.2 | 42510.4 | 5217.9 | 4260.2 | 43953.6 | 3835.4 | 2696.8 | 32700.8 | 2228.6 | 1649.7 |

"-SIMD/thread" indicates enabled technique. The results revealed that TF.js's Wasm backend offers limited model support, accommodating only four models. In the vanilla Wasm backend, the prediction latency of the four supported models averages 5105.2 ms. By activating SIMD and multithreading, the average prediction latency is 4260.2 ms and 2696.8 ms, respectively. The average prediction latency reduction of each model averages 36.2% and 13.3%, respectively. Furthermore, enabling both technologies concurrently led to an average decrease in prediction latency by 54.1%, where the latency of all models averages 1649.7 ms, highlighting their combined effectiveness in accelerating in-browser inference.

to the differences in framework implementation, as the two browsers use different engines, which accounts for the variations in performance.

In summary, the performance differences in performing in-browser inference between browsers are minimal, compared with native inference. Although Chrome's in-browser inference performance is slightly better than Firefox's, the gap is still much smaller than the difference between browser-based and native inference. This discrepancy is mainly due to variations in the internal implementations of the browsers. The slight differences between the two browsers are also because both are seasoned engineering experts who optimize both to support general JavaScript execution and browsing tasks.

*Results on Mobile Devices.* In addition, we explored the inference latency performance in-browser on mobile devices. We selected a specific model for each type of application and presented the results in Tables 8–11. The experiments were conducted using **TensorFlow Lite (TFLite)** and **ONNX Mobile Runtime (mORT)**.

Starting with the analysis of the results of the CPU. The first aspect is prediction latency, where we enabled SIMD and set up four threads. By comparing Table 8 with Tables 9 and 10, we found that the difference between native and browser environments on mobile devices is still significant as on PCs. Specifically, the latency using TF.js for prediction is 18.9× that of using TFLite, and for ORT.js and mORT, this difference is 14.5×. For both frameworks, the average gap reaches 15.8×. The difference between native and browser mainly stems from resource contention. This is primarily

Table 10. Latency (ms) of Models in ORT.js Wasm Backend on Mobile Devices

| | Wasm | | | Wasm-thread | | | Wasm-SIMD | | | Wasm-thread-SIMD | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction | setup | warmup | prediction |
| (M1) MobilenetV2 | 4622.5 | 442.5 | 408.1 | 6605.8 | 384.8 | 276.6 | 6373.1 | 350.1 | 254.9 | 5992.8 | 324.1 | 226.2 |
| (M2) Resnet50 | 15954.1 | 1571.0 | 1545.8 | 18448.7 | 1248.7 | 1186.3 | 18609.3 | 1213.1 | 1015.9 | 17918.3 | 1103.1 | 901.8 |
| (M3) VGG16 | 195834.1 | 4085.7 | 4229.6 | 204477.7 | 3546.4 | 2847.4 | 183901.6 | 3298.8 | 2490.5 | 191393.7 | 3013.0 | 2398.2 |
| (M4) SSD-MobilenetV2 | 23861.4 | 1172.4 | 1317.5 | 23468.6 | 956.9 | 927.7 | 23181.6 | 913.2 | 875.4 | 21861.1 | 854.7 | 779.1 |
| (M5) EfficientDet | 20333.2 | 6328.3 | 5689.6 | 18789.1 | 4361.7 | 4080.2 | 18469.7 | 4073.2 | 3589.3 | 18689.6 | 4926.0 | 3430.4 |
| (M6) Yolo5-Middle | 4928.9 | 2018.6 | 1669.9 | 4700.4 | 1577.6 | 1170.3 | 4568.2 | 1354.1 | 1026.5 | 4547.6 | 1145.8 | 994.8 |
| (M7) Bert-base | 24412.7 | 1515.9 | 1365.7 | 22893.9 | 920.9 | 961.8 | 22875.6 | 960.8 | 754.6 | 21673.4 | 825.4 | 732.4 |
| (M8) MobileBert | 19406.6 | 1586.2 | 1746.0 | 20109.7 | 1226.8 | 1215.1 | 19340.8 | 1140.8 | 1090.0 | 18144.7 | 1070.8 | 966.6 |
| (M9) Bert-Small | 6550.4 | 89.9 | 97.4 | 5820.4 | 78.6 | 73.7 | 5667.4 | 70.5 | 61.9 | 5802.1 | 63.4 | 58.7 |
| Average | 35100.4 | 2090.1 | 2007.7 | 36146.0 | 1589.2 | 1415.5 | 33665.3 | 1486.1 | 1239.9 | 34002.6 | 1480.7 | 1165.4 |

"-SIMD/thread" indicates enabled technique. The results revealed that ORT.js's Wasm backend is fully compatible, supporting all nine models. In the vanilla Wasm backend, the average prediction latency of the models is 2007.7 ms. Significant reductions in prediction latency were observed after enabling SIMD and multithreading, where the average prediction latency is 1239.9 ms and 1415.5 ms, respectively. For each model, the prediction latency decrease of each model averages 37.8% and 17.0%, respectively. Enabling both technologies further reduced the prediction latency to 1165.4 ms, and the prediction latency decrease of each model averages 42.4%.

Table 11. Latency (ms) of Models in WebGL Backend on Mobile Devices

| | ORT.js-WebGL | | | TF.js-WebGL | | |
|---|---|---|---|---|---|---|
| | Setup | Warmup | Prediction | Setup | Warmup | Prediction |
| (M1) MobileNetV2 | 6934.2 | 1424.6 | 66.8 | 7962.4 | 8986.7 | 47.8 |
| (M2) ResNet50 | 80720.6 | 3559.6 | 456.3 | 33607.8 | 8284.6 | 626.3 |
| (M3) VGG16 | 198767.6 | 17329.5 | 1102.2 | 154695.5 | 7281.8 | 1892.0 |
| (M4) SSD-MobileNetV2 | / | / | / | 17363.4 | 17247.1 | 1223.2 |
| (M5) EfficientDet-D1 | / | / | / | 34967.2 | 18185.6 | 3712.5 |
| (M6) Yolo5-Middle | / | / | / | 30862.0 | 21188.9 | 374.2 |
| (M7) Bert-base | / | / | / | 47464.7 | 5313.2 | 984.5 |
| (M8) MobileBert | / | / | / | 47378.7 | 3528.3 | 1136.4 |
| (M9) Bert-Small | / | / | / | 12449.0 | 4233.5 | 192.6 |
| Average | 95474.1 | 7437.9 | 541.8 | 42972.3 | 10472.2 | 1132.2 |

"/" indicates the same as Table 4. The average prediction latency of TF.js in the WebGL backend is 541.8 ms; the average prediction latency of ORT.js is 1132.2 ms. The results indicate that TF.js's WebGL backend supports more models, supporting six more models than ORT.js, the same as the results on P C devices. Additionally, we noted that the warmup latency for GPU inference exceeds the prediction latency, with an average increase of 34.6× and 15.0× on average, for TF.js and ORT.js, respectively.

because mobile devices inherently have limited resources, and in-browser inference also requires browser support. Running Chrome on mobile devices inevitably leads to resource competition. When performing native inference, we directly utilized the C++ implementation and executed the model inference via "adb shell," thus significantly avoiding resource contention. In contrast, in-browser inference relies on the Chrome browser. Additionally, the absence of FMA instructions in Wasm also impacts performance. The second aspect is warmup latency. In the mobile environment, the latency for TF.js is 25.7× of TFLite, while the latency of ORT.js is 8.0× of mORT. For both frameworks, the average warmup latency gap between the native and browser environment is 10.2×. Additionally, we observed that compared to TFLite, mORT also demonstrated lower warmup latency, a result that is similar to what we see on the PC side. Compared to prediction latency, the warmup latency for TF.js is 1.3× that of the prediction latency, while for ORT.js, this gap is 1.2×. The difference in latency between the two stages is not as significant as that between the browser and the native environment. The variation between the two stages mainly arises from additional overheads such as memory allocation during the first model prediction. The final aspect

is setup latency. We found that the latency for TF.js is 117.2× of TFLite; for ORT.js, its latency is 28.1× of mORT. For both frameworks, the average setup latency gap is 62.0×. We discovered that this gap is much larger than that on PC devices. Compared to warmup and prediction, setup latency is mainly affected by model size, library size, and network conditions. The setup latency of in-browser inference is high because mobile devices can only connect to wireless networks. According to our measurement, the download bandwidth ranges from 5.4 MB/s to 7.8 MB/s, which is much slower than that on PC devices using high-speed wired networks. Different from the PC side, on mobile devices both libraries are lightweight. Specifically, the TFLite library is only 5.3 MB, and the mORT library is just 14 MB. This indicates that the latency for loading libraries on mobile devices is low. However, as the model changes, there are significant differences in the latency for loading models. For example, the setup latency for MobileNetV2 is only 22.0 ms, while for VGG16 it reaches 668.6 ms. This difference is primarily because the setup stage also includes loading the model. During this process, the framework loads the model to memory, involving parsing the model structure, and thus, as the model structure becomes more complex, the setup latency increases accordingly.

Furthermore, we analyzed the impact of different acceleration technologies, i.e., multithreading and SIMD. As with previous experiments, we set four threads when enabling multithreading. We found that SIMD could reduce the prediction latency for TF.js by 36.2% and for ORT.js by 37.8%, while multithreading could reduce the latency by 13.3% and 17.0% for TF.js and ORT.js, respectively. When both technologies were enabled, prediction latency decreased by 54.1% and 42.4% for TF.js and ORT.js, respectively. Similar to PC devices, SIMD also brings more acceleration than multithreading on mobile devices, and the effect of multithreading is not as pronounced as on PCs due to limited resources on mobile devices, leading to significant resource competition when four threads are enabled, thus lowering the multithreading performance.

Next, we analyzed data on the GPU. The first aspect is prediction latency. In the mobile environment, the latency for TF.js is 9.4× of TFLite; for ORT.js, its latency is 5.6× of mORT. The average gap for both frameworks is 7.8×. This difference is much smaller than on PC devices, mainly because mobile GPUs do not possess the powerful parallel computing capabilities of Nvidia GPUs and do not have advanced libraries like CUDA for PCs, hence the smaller difference. In-browser inference is slower than native inference, primarily because kernel implementation of native inference is implemented directly via OpenCL kernels, which can run on mobile devices directly after compilation. In contrast, in-browser inference requires implementation through WebGL. The second aspect is warmup latency. In the mobile environment, the latency for TF.js is 242.2× of TFLite on average; the latency of ORT.js is 29.2× of mORT on average. For both frameworks, the average gap between the mobile native and browser environment is 162.3×. Such larger gaps mainly arise from the poorer performance of mobile GPUs, and in-browser WebGL shader code compilation depends on GPU performance, thus the significant difference. The gap on mobile devices compared to the gap on PC GPUs is quite different, primarily due to the use of completely different hardware. On PCs, due to limitations of the native inference framework, we use discrete GPUs and perform inference with CUDA, which involves considerable initialization latency for the discrete GPUs, whereas on mobile devices, the GPU is on the same chip as the CPU. Additionally, TFLite's lack of support for some models due to framework implementation also impacts the results. Nevertheless, there is still a large gap between the warmup latencies of native inference and in-browser inference on mobile devices. The last aspect is setup latency. In the mobile environment, the latency for TF.js is 21.7× of TFLite; for ORT.js, it is 315.8× of mORT. For both frameworks, the average setup latency gap is 81.7×. This difference is also larger compared to PCs. The setup stage of in-browser inference is heavily influenced by network conditions, as analyzed in the CPU result part. Mobile devices suffer from higher setup stage latency due to lower network transmission bandwidth, a

problem not present in native inference. Native inference merely requires loading from the local file system.

In summary, on mobile devices, the performance differences between native and browser environments are still large, primarily due to the constraints of SIMD on CPUs, and inevitable resource competition. Compared to PC devices, in-browser inference latency on mobile devices is generally higher, largely due to limited resources. This scenario highlights both advantages and disadvantages. On the positive side, the uniformity between native and browser environments simplifies the development and deployment processes of in-browser inference. However, the lack of powerful computational libraries and resource constraints can lead to increased latency, presenting a significant challenge for real-time applications on mobile platforms.

*4.1.2 Kernel-Level Latency.* We explore the kernel-level inference performance of in-browser inference. Due to the immature support for profiling in ORT.js, we use TF.js for analysis. We provide prediction results in Table 12 and warmup results in Table 13. The setup stage is excluded because kernels are not executed in this stage.

First, we explore which kernels dominate the *prediction latency*. Both multithreading and SIMD are enabled and we set four threads in the Wasm backend. Each percentage value in Table 12 shows the latency proportion out of the total latency in a setting. We find that computation-intensive kernels predominantly contribute to the overall latency in both backends, although the number of memory-intensive kernels makes up 37.7% of the total kernels. Specifically, Einsum and FusedConv2d contribute more than 70% of the total latency. We also find that the top five kernels are different in the two backends due to different computation and memory access patterns. In the Wasm backend, memory-intensive kernels introduce more latency. For example, GatherV2 accounts for the third most prediction latency in Wasm (12.5%), but it is not in the top five kernels in WebGL. The reason is that interthread synchronization relies on message passing in Wasm [43], which introduces additional overhead and becomes particularly pronounced for memory-intensive kernels. These results indicate that computation-intensive kernels remain a focal point for optimization in both backend kernels.

We also explore the impact of SIMD and multithreading on the kernel during the prediction stage. We set four threads when enabling multithreading. We compared the latency of each kernel with various acceleration technologies enabled, compared to the latency in the vanilla Wasm setting, and represented these variations using different cell colors in Table 12. Firstly, we found that enabling multithreading led to an increase of up to 3.7% in the prediction latency of memory-intensive kernels compared to the settings without multithreading enabled. Furthermore, the proportion of latency contributed by these kernels within the overall inference process increased by up to 8.6% (as seen in the results for GatherV2 in the "Wasm-S" and "Wasm-S-T" settings). This is because these memory-intensive kernels have limited support for multithreading. We dive into the kernel implementation in the source code and find that multithreading is only employed in some kernels, including Pad, among others. When enabling SIMD, we find that the latency proportion of computation-intensive kernels decreases by up to 6.8% and memory-intensive kernel latency changes negligibly (as indicated by the cell color). Additionally, we observed that the proportion of computation-intensive kernel latency within all kernels decreased by up to 12.8% (as seen in the results for Einsum in the "Wasm-T" and "Wasm-S-T" settings). This is because Wasm has limited support for data-moving SIMD instruction, hindering the potential acceleration for memory-intensive kernels through SIMD. When computation-intensive kernels' latency decreases and memory-intensive kernels' latency remains unchanged, the proportion of latency attributed to memory-intensive kernels also increases. Conversely, the reason for the decrease in the proportion of latency for computation-intensive kernels is the same. Nevertheless,

Table 12. Kernel Prediction Latency Proportion in TF.js

| Kernel | Wasm | Wasm-S | Wasm-T | Wasm-S-T | Kernel | WebGL-I | WebGL-D |
|---|---|---|---|---|---|---|---|
| Einsum | 48.2% | 46.5% | 48.0% | 41.4% | Einsum | 37.1% | 48.4% |
| FusedConv2D | 47.6% | 44.3% | 47.4% | 34.6% | FusedConv2D | 43.6% | 23.7% |
| GatherV2 | 1.3% | 3.9% | 1.5% | 12.5% | Add | 1.2% | 1.6% |
| Slice | 0.3% | 1.1% | 0.3% | 2.8% | Multiply | 1.8% | 1.4% |
| FusedMatMul | 0.8% | 1.8% | 0.8% | 1.6% | FusedMatMul | 2.3% | 2.9% |
| FusedDwConv2D | 1.3% | 1.2% | 1.3% | 1.9% | FusedDwConv2D | 1.5% | 2.1% |

"-S/T" indicates SIMD/multithreading. "Dw" indicates "Depthwise." "-I/D" indicates integrated/discrete GPU. Each percentage value represents the proportion of total latency for the kernel under the setting, such as Wasm-S which indicates enabling SIMD in Wasm. For example, GatherV2 accounts for 3.9% of the total latency in the Wasm-S setting. The cell color represents the change in latency for the same kernel when different technologies are enabled on top of the vanilla Wasm. Cell color gray means the latency is a little longer than that of Wasm (1% < increment < 5); cell yellow means the latency is close to that of Wasm (difference <1%); cell blue means the latency is much less than that of Wasm (decrement >10%). For example, the yellow cell color for GatherV2 indicates that enabling SIMD in Wasm leads to less than a 1% difference in the prediction latency of GatherV2 compared to the vanilla Wasm backend. The percentages and cell colors provide independent information. The data indicate that across all settings in the Wasm backend and GPU, computation-intensive kernels consistently dominate the in-browser inference prediction latency. However, due to the synchronization overhead among threads, the Wasm backend exhibits a higher proportion of memory-intensive kernels compared to the WebGL backend.

each technique effectively accelerates computation-intensive kernels. For instance, compared with vanilla Wasm, SIMD and multithreading reduce the average prediction latency of FusedConv2D by 86.9% and 66.7%, respectively. When applying both techniques, the prediction latency decreases by 89.9%.

During the warmup stage, our observations remain consistent with previous findings. The latency in both backends is predominantly influenced by computation-intensive kernels. However, the Wasm backend exhibits a higher latency contribution from memory-intensive kernels compared to the WebGL backend. Upon enabling SIMD, we note a reduction in the latency proportion attributable to computation-intensive kernels by up to 2.0%. This reduction is smaller than that observed in the prediction stage, primarily because memory allocation incurs additional overhead that is unaffected by SIMD. Although the computation time for computation-intensive kernels decreased, extra overhead is added due to memory allocation, which remains constant. Moreover, the latency of memory-intensive kernels also does not significantly decrease; these result in a modest reduction in the proportion of computation-intensive kernels after SIMD is enabled. Conversely, when multithreading is enabled, the latency proportion of memory-intensive kernels experiences an increase of up to 3.2%. This trend mirrors the prediction stage, where multithreading offers negligible acceleration to memory-intensive kernels while introducing extra overhead.

In summary, by analyzing the data from Tables 12 and 13, we find that computationally intensive kernels, such as Einsum and FusedConv2D, account for at least 76.0% of total latency. Within the Wasm backend, SIMD has almost no effect on memory-intensive kernels, while multithreading, in contrast, leads to an increase in memory-intensive kernel latency. In contrast, both technologies reduce the latency of computationally intensive kernels by more than 10%. Upon diving into the kernel implementations, we find that this is primarily due to insufficient multithreading optimizations for memory-intensive kernels by the framework. In the WebGL backend, computation-intensive kernels similarly dominate the latency. Therefore, for both backends, improving the latency of computationally intensive kernels has a greater impact on overall latency, and framework vendors should also focus on optimizing multithreading for memory-intensive kernels.

Table 13. Kernel Warmup Latency Proportion in TF.js

| Kernel | Wasm | Wasm-S | Wasm-T | Wasm-S-T | Kernel | WebGL-I | WebGL-D |
|--------|------|--------|--------|----------|--------|---------|---------|
| Einsum | 36.6% | 33.5% | 35.0% | 33.4% | Einsum | 38.1% | 36.7% |
| FusedConv2D | 54.1% | 52.3% | 53.4% | 52.1% | FusedConv2D | 49.5% | 52.5% |
| GatherV2 | 2.6% | 3.1% | 2.4% | 4.5% | Add | 1.8% | 2.6% |
| Slice | 0.6% | 1.2% | 2.9% | 3.8% | Multiply | 2.4% | 3.4% |
| FusedMatMul | 2.4% | 2.1% | 1.4% | 1.3% | FusedMatMul | 1.9% | 2.0% |
| FusedDwConv2D | 2.1% | 1.3% | 1.1% | 1.1% | FusedDwConv2D | 3.1% | 2.8% |

The table format remains the same with Table 12. We found that similar to the prediction stage, computation-intensive kernels dominate the latency during the warmup stage. Due to the overhead of synchronization among threads in the Wasm backend, memory-intensive kernels introduce a higher latency proportion compared to those in the WebGL backend.

> *Finding 1*: A large inference performance disparity exists between browser and native environments. The prediction latency gap in CPU is 16.9× on average. The shorter SIMD instructions and Wasm VM-inherent inefficiency are key reasons for the low performance of the Wasm backend. SIMD provides up to 50.7% latency reduction, and multithreading provides up to 31.2% latency reduction. WebGL significantly accelerates prediction by up to 4.9× compared to Wasm but suffers from severe cold start issues, with a slowdown of up to 64.6×, mainly due to in-browser shader compilation. The prediction latency spectrum across devices reaches up to 28.4× in Wasm and 19.4× in WebGL, which is mainly attributed to different hardware capacities. Memory-intensive kernels can contribute up to 12.5% of total latency.

*Kernel-Level Latency Difference across Different Models and Tasks.* To explore the differences in latency across different kernels, we primarily examined the types of kernels included in various models and the relationship between the **Floating Point Operations Per Second (FLOPs)** of these kernels and their corresponding latencies. We focused on three specific operators: Einsum, FusedDwConv2D, and FusedConv2D. FusedDwConv2D is mainly used in the MobileNetV2 and SSD-MobileNetV2 models, while FusedConv2D is commonly found in other convolutional neural networks. Einsum, primarily used for matrix multiplication, is utilized across all models. For instance, FusedConv2D accounts for 87.4% prediction latency of ResNet50 in the Wasm backend with both multithreading and SIMD enabled, while Einsum accounts for less than 0%. In contrast, Einsum accounts for 92.1% prediction latency of MobileBert in the same setting while FusedConv2d accounts for 0%. We found that models utilizing FusedDwConv2D, typically used in classification and detection tasks, exhibit both lower FLOPs and latency. Specifically, FusedDwConv2D accounts for 0.6% parameters and 1.9% prediction latency. Although this may sacrifice some inference accuracy, it significantly reduces latency. Running more lightweight models in browsers holds potential, as these models are computationally efficient. In summary, all three kernels are computation-intensive kernels that dominate the total latency, as well as the number of parameters. Specifically, the three kernels account for 99% of total parameters. These results indicate that optimization on these kernels can significantly improve the in-browser inference performance, i.e., reducing latency.

*Results on Firefox Browser.* We also explored results on Firefox, presenting the results of prediction and warmup latencies in Tables 14 and 15, respectively. Starting with the prediction stage, the results are similar to those in Chrome. Computationally intensive kernels, such as FusedConv2d and Einsum, still represent the largest latency proportion even after enabling multithreading and SIMD, accounting for 41.6% and 34.1%, respectively. This is primarily because the prediction stage of model inference is still dominated by computation. Upon separately enabling multithreading

Table 14.   Kernel Prediction Latency Proportion in TF.js in Firefox Browser

| Kernel | Wasm | Wasm-S | Wasm-T | Wasm-S-T | Kernel | WebGL-I | WebGL-D |
|---|---|---|---|---|---|---|---|
| Einsum | 47.4% | 46.2% | 47.8% | 41.6% | Einsum | 38.2% | 48.1% |
| FusedConv2D | 46.9% | 43.2% | 45.4% | 34.1% | FusedConv2D | 44.4% | 41.6% |
| GatherV2 | 1.4% | 3.0% | 1.8% | 10.9% | Add | 1.4% | 1.9% |
| Slice | 0.4% | 1.1% | 0.6% | 3.6% | Multiply | 1.7% | 1.3% |
| FusedMatMul | 0.9% | 0.9% | 0.8% | 2.4% | FusedMatMul | 2.5% | 2.6% |
| FusedDwConv2D | 1.2% | 1.3% | 1.4% | 1.8% | FusedDwConv2D | 3.1% | 1.4% |

The table format remains the same with Table 12. The results show that in all settings and hardware environments, the prediction latency for in-browser inference is predominantly influenced by computationintensive kernels, such as FusedConv2D. Additionally, the Wasm backend demonstrates a greater prevalence of memory-intensive kernels due to the overhead associated with thread synchronization, in contrast to the WebGL backend.

Table 15.   Kernel Warmup Latency Proportion in TF.js in Firefox Browser

| Kernel | Wasm | Wasm-S | Wasm-T | Wasm-S-T | Kernel | WebGL-I | WebGL-D |
|---|---|---|---|---|---|---|---|
| Einsum | 35.9% | 33.0% | 34.8% | 33.2% | Einsum | 39.2% | 36.9% |
| FusedConv2D | 54.5% | 51.6% | 52.8% | 52.3% | FusedConv2D | 49.1% | 51.4% |
| GatherV2 | 2.9% | 3.3% | 2.6% | 4.1% | Add | 1.9% | 2.4% |
| Slice | 0.8% | 1.7% | 3.3% | 4.0% | Multiply | 2.1% | 3.6% |
| FusedMatMul | 2.1% | 2.0% | 1.5% | 1.5% | FusedMatMul | 1.8% | 2.1% |
| FusedDwConv2D | 2.0% | 1.7% | 1.1% | 1.2% | FusedDwConv2D | 2.9% | 2.4% |

The table format remains the same with Table 12. We observed that during the warmup stage, the latency of computation-intensive kernels remains the predominant factor. Moreover, due to the additional overhead from thread synchronization in the Wasm backend, the latency proportion of memoryintensive kernels exceeds that of the WebGL backend.

and SIMD, we observed a significant reduction in latency for these computationally intensive kernels compared to vanilla Wasm. Specifically, SIMD reduced the latency of computation-intensive kernels by 84.5%, and multithreading reduced it by 63.1%. However, due to the presence of memory-intensive kernels and the framework's lack of multithreading support for these kernels, along with limited acceleration from SIMD, the absolute latency values of memory-intensive kernels remained essentially unchanged, while their latency proportion increased when both technologies were enabled. On the GPU side, we arrived at similar conclusions. Due to the absence of thread synchronization and message passing, the latency proportion of memory-intensive kernels is so as low that they do not appear in the table.

Regarding the warmup stage, shown in Table 15, we found that computation-intensive kernels continue to dominate the latency. However, memory-intensive kernels have a higher latency proportion in the Wasm backend. We also noted that compared to the vanilla Wasm, the proportion of computation-intensive kernels is lower during the warmup stage when acceleration techniques are enabled. For instance, under conditions enabling both SIMD and multithreading, the latency proportion of FusedConv2D and Einsum decreased by 2.2% and 2.7%, respectively. This stage includes memory allocation, which is not affected by hardware performance. Similar to the prediction stage, multithreading technology, due to the overhead of thread synchronization, actually increased the latency of memory-intensive kernels (as indicated by the gray cell color in the tables). In contrast, the WebGL backend does not face such issues, maintaining a consistent dominance of computation-intensive kernels in terms of warmup latency.

In summary, the data on Firefox closely mirror the results on Chrome, aligning with our previous analysis. The changes in kernel latency proportions and absolute values primarily depend on the model architecture and the implementation of the framework, independent of the browser. This

Table 16. Kernel Prediction Latency Proportion in TF.js on Mobile Devices

| Kernel | Wasm | Wasm-S | Wasm-T | Wasm-S-T | Kernel | WebGL |
|---|---|---|---|---|---|---|
| Einsum | 43.4% | 41.9% | 42.9% | 38.2% | Einsum | 41.3% |
| FusedConv2D | 51.1% | 47.8% | 49.9% | 43.1% | FusedConv2D | 45.1% |
| GatherV2 | 1.2% | 3.9% | 2.5% | 9.8% | Add | 1.4% |
| Slice | 0.6% | 1.7% | 1.3% | 4.1% | Multiply | 1.0% |
| FusedMatMul | 0.9% | 1.8% | 1.0% | 2.0% | FusedMatMul | 3.7% |
| FusedDwConv2D | 1.8% | 2.2% | 1.7% | 1.9% | FusedDwConv2D | 2.8% |

The table format remains the same with Table 12. The results reveal that computation-intensive kernels primarily drive the prediction latency in in-browser inference during the prediction stage in all scenarios. For instance, the latency of FusedConv2D and Einsum accounts for more than 81%. Memoryintensive kernels do not introduce much latency proportion in the WebGL backend compared to the Wasm backend, because of the need for thread synchronization in the Wasm backend, especially when multithreading is enabled.

Table 17. Kernel Warmup Latency Proportion in TF.js in Firefox Browser

| Kernel | Wasm | Wasm-S | Wasm-T | Wasm-S-T | Kernel | WebGL |
|---|---|---|---|---|---|---|
| Einsum | 40.1% | 36.2% | 37.8% | 35.1% | Einsum | 42.5% |
| FusedConv2D | 50.5% | 46.1% | 47.6% | 45.3% | FusedConv2D | 48.3% |
| GatherV2 | 3.1% | 4.3% | 5.1% | 8.5% | Add | 1.4% |
| Slice | 1.2% | 1.9% | 3.9% | 4.0% | Multiply | 1.4% |
| FusedMatMul | 2.4% | 3.1% | 3.0% | 3.2% | FusedMatMul | 0.9% |
| FusedDwConv2D | 2.1% | 1.9% | 1.8% | 2.0% | FusedDwConv2D | 1.1% |

The table format remains the same with Table 12. On mobile devices, computation-intensive kernels dominate the warmup total latency, with the latency of FusedConv2D and Einsum exceeding 80% in all scenarios. Additionally, in the Wasm backend, the impact of memory-intensive kernel latency becomes significant upon enabling multithreading, resulting in a latency proportion of 12.5% for GatherV2 and Slice.

is consistent with Wasm's browser-independent mechanism and corroborates our model-level conclusions in the Chrome browser.

*Results on Mobile Devices.* We also explored the results on mobile devices, displaying the prediction and warmup latencies in Tables 16 and 17, respectively. Starting with the prediction stage, shown in Table 16, it was unsurprising that computationally intensive kernels continued to have the highest latency proportion, consistent with previous conclusions. After enabling SIMD, the proportions of FusedConv2d and Einsum decreased by 3.3% and 1.5%, respectively, mainly due to the significant acceleration SIMD brings to these computation-intensive kernels. However, SIMD provides limited acceleration for memory-intensive kernels. After enabling multithreading, we observed an increase in the proportions and the absolute latency values of these computation-intensive kernels by 2.0% and 4.1%, respectively. When both technologies were enabled, all computation-intensive kernels saw their largest proportion decrease, with FusedConv2d and Einsum reducing by 8.0% and 5.2%, respectively. These results align with the conclusions drawn from both browsers on the PC.

Turning to the warmup stage results in Table 17, the proportion of latency attributed to memory-intensive kernels is higher in the Wasm backend compared to the WebGL backend, reaching 12.5% after enabling SIMD and multithreading. This is mainly due to the limited optimization of these kernels by the framework when utilizing both acceleration technologies. The cost of memory allocation leads to a decrease in the proportion of computation-intensive kernels compared to the prediction stage, primarily because CPU capacity has minimal impact on memory allocation; this essentially adds a fixed value to the computational latency, thus reducing the proportion of

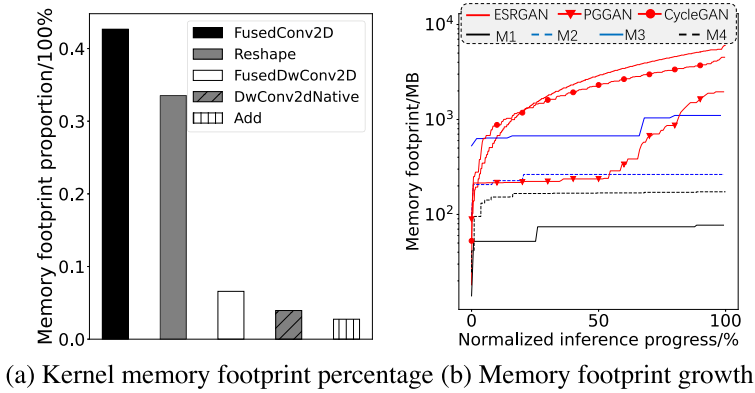(a) Kernel memory footprint percentage (b) Memory footprint growth

Fig. 5. Memory footprint analysis. We found that: (a) Memory-intensive kernels can occupy a significant memory footprint. For example, the memory requirements for Reshape are second only to FusedConv2D, even though their latency proportions are quite low. (b) In most cases, inference memory tends to stabilize as the warmup progresses. However, for super-resolution models, the inference memory footprint continues to grow because the size of the intermediate data generated during inference gradually increases.

computationally intensive kernels during the warmup stage. In the WebGL backend, there is no additional synchronization overhead like that in Wasm multithreading; hence, the proportion of memory-intensive kernel latency remains low so that the memory-intensive kernels do not appear in the table.

In summary, the results on mobile devices are consistent with those observed on PCs using Chrome and Firefox. The analysis involves only the framework and kernel implementation and optimization in two backends; thus, conclusions and analyses applicable to PCs are equally relevant to mobile devices.

## 4.2   Memory Footprint Analysis

We use the "window.performance.memory" JavaScript API to approximately measure the memory footprint. This API provides an approximation of a Web page's memory usage. We limit our analysis to the Wasm backend of TF.js for the following reasons: (1) After delving into the framework's implementation, we find that the memory access patterns are consistent across both backends within the same framework. (2) The WebGL backend presents challenges in accurately profiling the memory footprint at the kernel level on the GPU side. This is because the API measures the JavaScript heap size, which is indicative of the memory footprint on the CPU, whereas GPU memory allocations are handled separately. (3) In the case of the Wasm backend, while ORT.js runs all logic, including kernel scheduling and execution, entirely in Wasm, this JavaScript API cannot execute within Wasm. We also lacked a reliable method to precisely capture the kernel memory footprint from the Wasm. In contrast, TF.js implements the scheduling logic in JavaScript, which subsequently invokes the Wasm kernels for computation, allowing us to use this API to effectively and accurately profile the kernel memory footprint. Due to these considerations, we focus on the Wasm backend of TF.js in our analysis. It is important to note that SIMD and multithreading, which are enabled by default, do not influence the memory footprint. We measure the top five kernels that dominate the memory footprint during inference and illustrate the results in Figure 5(a). We find that memory-intensive kernels, such as *Reshape*, consume a significant amount of memory during inference, yet exhibit low latency.

To further understand the memory access patterns of different models, we illustrate the memory growth during warmup in Figure 5(b) because the framework only allocates memory in this stage. We involve three super-resolution models in this experiment, i.e., ESRGAN [114], PGGAN [84] and CycleGAN [124]. These models are excluded from the latency analysis because ORT.js does not support them in both backends. We find that the curves of the models used in latency analysis show a similar pattern, i.e., initially increasing and then stabilizing. This is because the framework retains the allocated memory without releasing it back to the operating system during warmup. Specifically, DL frameworks always manage allocated memory themselves to avoid the overhead of system calls. During the warmup stage, the framework requires memory from the operating system when self-managed free memory blocks are not enough, which is then consistently used for subsequent inferences. The blocks are released at the end of the inference process. This approach optimizes memory usage by reusing allocated memory and enhancing overall performance. The memory footprint of the super-resolution model inference grows continuously. Take ESRGAN inference as an example, the peak memory footprint of the entire process spikes to 6,500 MB, even though the ESRGAN model itself is only 19 MB in size. This means that the memory required for the entire inference process is 334.6× greater than that of the model itself, despite the batch size being 1. This is attributed to the expansion of intermediate data size, surpassing the size of previously allocated memory blocks, leading to additional requests for larger memory blocks from the operating system and a continuous increase in memory footprint. Such results emphasize the importance of efficient memory management design, as excessive memory footprint can hinder the ability for model inference.

*Results on Firefox Browser and Mobile Devices.* In the course of our analysis of DL inference memory footprint, we have considered extending our study to encompass data from Firefox and mobile devices. However, there are a couple of considerations that have led us to refrain from a detailed analysis in these areas. Firstly, on Firefox, the "performance.memory" API is unfortunately not available, and currently, we have not identified an alternative method that meets our needs for accurate memory usage assessment. Secondly, our preliminary observations of the memory usage data on mobile devices showed that the results are identical to those on PCs. This is expected since the memory footprint is only determined by the implementation of the framework, which is consistent across different devices. Given this uniformity and the lack of additional insights that new data might provide, we exclude these parts of the data in our analysis.

In summary, during model inference, there are numerous memory-intensive kernels. Although these kernels have low latency, they impose much memory pressure. DL frameworks manage their own memory allocation to reduce system call overhead. However, this approach may not be efficient. For instance, in super-resolution models, increasing intermediate data leads to continuous memory growth during inference. Therefore, more sophisticated memory management mechanisms, such as profile-based memory pool [113], are necessary. Additionally, we observed that memory footprint is solely related to model structure and framework implementation, independent of hardware, devices, or browser platforms.

> *Finding 2*: Memory footprint can become a significant bottleneck during inference. For example, when performing super-resolution model inference like ESRGAN, the growing size of intermediate data can lead to a memory footprint surge of over 6,500 MB, which is 334.6× of the model size. Inference of such models surpasses the capacity of most devices, necessitating the need for more effective memory management techniques in frameworks. The framework can utilize the model's profile information and allocate a large enough memory block to meet peak memory footprint requirements.

Table 18. Responsiveness of Web Page

| | M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 |
|---|---|---|---|---|---|---|---|---|---|
| Baseline | 142.0 (2.3) | 142.0 (2.3) | 142.0 (2.3) | 142.0 (2.3) | 142.0 (2.3) | 142.0 (2.3) | 142.0 (2.3) | 142.0 (2.3) | 142.0 (2.3) |
| TF.js-Wasm | 92.7 (7.5) | 89.5 (11.0) | 91.2 (7.0) | 112.0 (7.0) | / | / | / | / | / |
| ORT.js-Wasm | 92.0 (5.0) | 94.5 (9.2) | 88.4 (9.8) | 108.0 (11.0) | 92.6 (12.0) | 91.0 (7.0) | 92.1 (6.6) | 115.0 (12.2) | 83.7 (7.0) |
| TF.js-WebGL-I | 87.6 (12.0) | 84.2 (7.5) | 94.2 (8.3) | 113.0 (13.0) | 100.4 (9.3) | 97.4 (11.2) | 89.3 (9.5) | 106.0 (7.7) | 83.1 (12.5) |
| TF.js-WebGL-D | 118.1 (2.9) | 119.2 (3.8) | 114.5 (3.6) | 127.4 (7.4) | 121.3 (6.3) | 112.0 (2.4) | 113.5 (3.1) | 120.8 (3.6) | 95.5 (8.6) |
| ORT.js-WebGL-I | 89.7 (7.7) | 88.2 (8.4) | 96.2 (4.4) | / | / | / | / | / | / |
| ORT.js-WebGL-D | 123.1 (5.1) | 119.7 (6.2) | 125 (4.0) | / | / | / | / | / | / |

Each entry is the average responsiveness score and the value in the bracket is the standard deviation of the score. "-D/-I," "/," and "M*" keep the same as Table 4. We found that in-browser inference can negatively impact responsiveness, reducing the responsiveness score by up to 45.4%. We found that the highest scores occur on discrete GPUs, due to their superior capacity.

## 5 Results of QoE Measurement

### 5.1 Responsiveness Analysis

The responsiveness QoE results are listed in Table 18. Without inference, the benchmark yields a mean score of 142.0, with a standard deviation of 2.3. When the DL is deployed, the mean score experiences a drop of up to 45.4% for both frameworks, while the standard deviation increases up to 5.7×. The degradation primarily stems from resource occupancy. Since the Speedometer benchmark performs tasks on both the CPU and GPU to simulate user interactions with Web pages, and model inference occupies at least one of the hardware resources, it inevitably leads to resource competition with the benchmark. This competition subsequently impacts the responsiveness scores. In terms of CPU and integrated GPU, the mean score difference is not significant. The mean scores for the integrated GPU are 1.8% lower than the CPU on average for TF.js, and 1.0% lower for ORT.js. Since the benchmark contains tasks that run on both the CPU and GPU, the execution of the benchmark tasks is influenced by the DL inference in either backend. Moreover, the parallelism capacity of the integrated GPU is relatively weak, thereby impacting the final scores. In contrast, the discrete GPU achieved the highest scores in all settings due to its outstanding performance. According to the GPU benchmark [38], there exists a maximum benchmark score gap of 207.4× between the discrete GPU and the integrated GPU in our dataset. We observed that the progress of benchmark tests slows down when inference and benchmark tasks run on the same hardware. These results indicate a tradeoff between responsiveness and inference performance. Achieving better responsiveness and low inference latency simultaneously becomes challenging. Timely responsiveness requires more resources to be allocated for processing requests, leaving fewer resources available for inference tasks. For TF.js and ORT.js, the responsiveness QoE results are similar across both frameworks. Using TF.js as the baseline, ORT.js responsiveness scores are, on average, 99.5% of TF.js on the Wasm backend. In the WebGL backend, on integrated and discrete GPUs, ORT.js responsiveness scores are 1.03× and 1.05× that of TF.js, respectively. This similarity is primarily because the responsiveness tests are only affected by resource competition, and both frameworks utilize similar resource configurations, such as multithreading and SIMD settings in Wasm. Therefore, the differences in responsiveness QoE resulting from using the two frameworks for inference are minimal.

Additionally, we compared the responsiveness QoE of different models across various tasks. In the image classification task, the VGG16 model scored the highest, averaging 101.6. In the object detection task, the SSD-MobileNetV2 model performed best, with an average score of 115.1. In the grammar checking task, the MobileBert model achieved the best scores, averaging 113.9. However, we did not find a correlation between prediction latency and responsiveness QoE. For example, VGG16 introduces the highest prediction latency while its responsiveness is the highest;

Table 19.  Smoothness of Web Page

|  | M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 |
|---|---|---|---|---|---|---|---|---|---|
| Baseline | 30.0 (0.00) | 30.0 (0.00) | 30.0 (0.00) | 30.0 (0.00) | 30.0 (0.00) | 30.0 (0.00) | 60.0 (0.00) | 60.0 (0.00) | 60.0 (0.00) |
| TF.js-Wasm | 23.8 (1.83) | 22.9 (1.79) | 22.4 (1.61) | 23.9 (1.65) | / | / | / | / | / |
| ORT.js-Wasm | 24.1 (1.79) | 23.1 (1.49) | 24.5 (1.71) | 23.8 (1.79) | 21.7 (1.66) | 22.4 (1.66) | 54.6 (4.07) | 53.7 (3.71) | 52.7 (3.51) |
| TF.js-WebGL-I | 11.2 (0.77) | 11.7 (0.85) | 12.9 (1.01) | 12.7 (0.82) | 11.1 (0.84) | 11.5 (0.79) | 26.8 (2.10) | 27.2 (1.82) | 23.5 (1.73) |
| TF.js-WebGL-D | 27.2 (1.29) | 26.4 (1.18) | 28.1 (1.51) | 28.3 (1.71) | 27.9 (1.26) | 28.1 (1.49) | 52.4 (2.44) | 51.5 (3.19) | 52.0 (2.85) |
| ORT.js-WebGL-I | 17.0 (1.11) | 16.8 (1.11) | 17.1 (1.31) | / | / | / | / | / | / |
| ORT.js-WebDL-D | 26.4 (1.39) | 23.0 (1.08) | 24.5 (1.14) | / | / | / | / | / | / |

Each entry is the average fps of the Web page/video and the value in the bracket is the standard deviation of fps across devices. "M*," "/," and "-D/-I" indicate the same as Table 4. We found that with inbrowser inference, fps decreased by up to 62.7%. Additionally, we found that fps performance on CPUs is better than on integrated GPUs, with an average fps reduction of 36.2% on CPUs compared to 44.3% on integrated GPUs. Discrete GPUs showed the best performance, with an average decrease of only 8.1%.

SSD-MobileNetV2 shows lower prediction latency than Yolo5-Middle but its responsiveness QoE is higher. We delved deeper into the distribution of test cases in the benchmark tests and found that CPU/GPU test cases alternated. In our experiments, inference and testing were also conducted alternately. We observed that if the WebGL backend was used during testing, the benchmark test would be blocked due to resource competition with WebGL. Testing would only resume when inference paused, and task switching within WebGL also incurred overhead. Therefore, the scores were affected by the distribution of inference and testing, but the fundamental cause was the resource competition introduced by inference.

In summary, in-browser inference decreases the responsiveness QoE scores, primarily due to resource competition. The Speedometer benchmark includes responsiveness test cases for both CPU and GPU; thus, regardless of the hardware used as the inference backend, the inference process competes with the responsiveness test for resources. Discrete GPUs, due to their superior capacity, demonstrate the best results in responsiveness QoE benchmark tests.

*Finding 3*: In-browser inference can significantly degrade the responsiveness by up to 45.4%. The reason is that the computational resources required by inference compete with other critical components of the Web page. Allocating more CPU and GPU resources to the DL-powered Web applications to minimize inference latency comes at the expense of responsiveness QoE.

### 5.2 Smoothness Analysis

The average fps QoE results are listed in Table 19. Additionally, Figure 6 illustrates the fps trace during SSD-MobileNetV2 inference in two backends in TF.js on a device (MacBook Pro 2020). Without inference, the average Web page rendering fps on the device is 60, and the average video playing fps is 30. When DL is deployed, the fps value inevitably drops (up to 62.7%). The integrated GPU exhibits the poorest performance in terms of smoothness, while the performance of the CPU is remarkably good, approaching that of the discrete GPU. Specifically, when performing DL model inference on the CPU, the fps experiences a relatively smaller drop, averaging 36.2% for both frameworks, compared to performance inference on the integrated GPU. Discrete GPU experiences 8.1% less fps degradation than the CPU and 44.3% less fps degradation than the integrated GPU. Furthermore, we observe that fps fluctuations are more severe when inferring on the integrated GPU, as depicted by the red dashed line in Figure 6. In Figure 6, the fps amplitude reaches 4, which is 31.5% of the average value (12.7). The fps amplitude can reach up to 37.3% of the average fps for all models. The reason is that rendering and video playing rely on GPU resources, leading to inevitable resource competition when performing inference in WebGL. The best performance

on discrete GPU is attributed to its high capacity. Moreover, in WebGL, the discrete GPU is not fully harnessed, allowing rendering and video-playing tasks to utilize the remaining capabilities. Additionally, we observed that during the setup stage, the fps is higher than the other stages. This is evidenced by the gray dashed line in Figure 6, where there is a sharp initial decline marking the setup stage. At this stage, there is an inflection point corresponding to an fps value of 27.2. Since the setup stage requires less CPU and GPU resources, the fps during this stage is comparatively higher than the other stages. However, due to the very low setup latency of the SSD-MobileNetV2, this segment occupies only a brief segment on the *x*-axis of Figure 6.

While performing inference on integrated GPU offers lower prediction latency, it may lead to fps degradation, because of its relatively lower capacity. On the contrary, performance inference on the CPU is slower but leads to less fps compromise since it alleviates resource competition. These results indicate the existence of a tradeoff between smoothness and inference performance. In terms of smoothness, the priority of using integrated GPU for inference is lower than other hardware.

Furthermore, we also compared the smoothness QoE of different models across tasks. In the image classification task, the MobileNetV2 model had the best fps, averaging 21.6. In the object detection task, the SSD-MobileNetV2 model achieved the highest fps, averaging 22.2. In the grammar checking task, the Bert-base model recorded the best fps, averaging 44.6. We observed that the fps differences among models within the same task are small. For example, during image classification tasks using the Wasm backend of TF.js, the fps difference between the three models was only 1.4. This was mainly because the smoothness experiments only involved rendering tasks, which were not blocked due to the resource competition from inference. Although the page rendering continued normally, it became less smooth instead of completely stalling, thus resulting in minimal fps differences among different models. However, regardless of the model, the inevitable resource competition always resulted in a decrease in smoothness QoE.

In summary, in-browser inference leads to significant resource contention, negatively impacting Web page rendering or video playing. Regardless of the framework or backend used, there is a noticeable drop in fps. Although WebGL offers lower inference latency, when inference is performed on integrated GPU, it competes with rendering tasks for GPU resources. In contrast, using Wasm as the inference backend results in a higher smoothness QoE. In our experiments, we found that discrete GPUs provide the best smoothness QoE, primarily because they are more powerful than other hardware options.

> *Finding 4*: In-browser inference impacts smoothness, with fps dropping by up to 62.7%. This is due to resource competition. The resource competition can lead up to 37.3% fps fluctuations. Achieving both smoothness and inference performance is a tradeoff. Using the Wasm backend for inference can lead to better smoothness compared to the WebGL backend on devices without a powerful discrete GPU.

## 5.3 Inference Accuracy Analysis

The average inference accuracy QoE results are listed in Table 20. In the ideal scenario where all images can be processed by the DL model, the classification accuracy is 17.2% and the detection mAP is 1.0. This lower classification accuracy is attributed to the dataset containing low-resolution images like 32×32 pixel size, and some frames contain multiple objects, which can also affect classification accuracy. The high detection mAP is mainly because the object region typically covers a significant portion of the image, and the dataset exclusively consists of images with a single class (face). This characteristic is well suited for in-browser detection tasks, like background blurring,
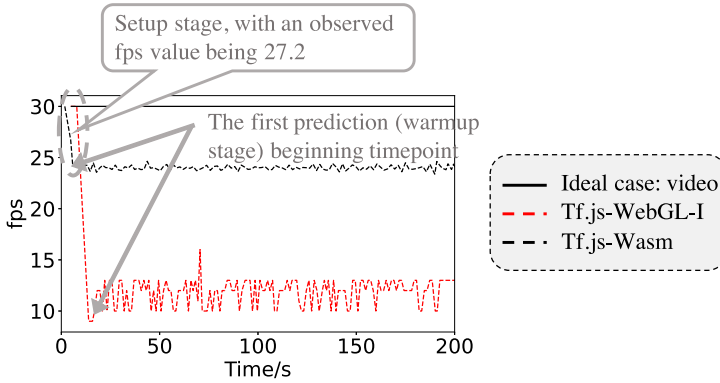
Fig. 6. Fps trace during SSD-MobileNetV2 inference. We observed that during in-browser inference, fps drops sharply and then continues to fluctuate, with the amplitude reaching 31.5% of the average value when inference is performed on the integrated GPU. The fps during the setup stage is higher than that during the prediction stage.

Table 20. Inference Accuracy on Certain Applications

| | M1 | M2 | M3 | M4 | M5 | M6 |
|---|---|---|---|---|---|---|
| Baseline | 17.2% (0.0) | 17.2% (0.0) | 17.2% (0.0) | 1.00 (0.0) | 1.00 (0.0) | 1.00 (0.0) |
| TF.js-Wasm | 13.7% ($0.53 \times 10^{-2}$) | 13.5% ($0.52 \times 10^{-2}$) | 8.8% ($0.33 \times 10^{-2}$) | 0.78 ($2.91 \times 10^{-2}$) | / | / |
| ORT.js-Wasm | 13.7% ($0.50 \times 10^{-2}$) | 13.6% ($0.52 \times 10^{-2}$) | 8.0% ($0.30 \times 10^{-2}$) | 0.78 ($3.10 \times 10^{-2}$) | 0.77 ($2.84 \times 10^{-2}$) | 0.79 ($3.00 \times 10^{-2}$) |
| TF.js-WebGL-I | 8.7% ($0.33 \times 10^{-2}$) | 8.8% ($0.32 \times 10^{-2}$) | 3.9% ($0.14 \times 10^{-2}$) | 0.76 ($2.86 \times 10^{-2}$) | 0.76 ($2.93 \times 10^{-2}$) | 0.78 ($2.83 \times 10^{-2}$) |
| TF.js-WebGL-D | 14.9% ($0.54 \times 10^{-2}$) | 14.2% ($0.52 \times 10^{-2}$) | 11.1% ($0.44 \times 10^{-2}$) | 0.95 ($3.26 \times 10^{-2}$) | 0.94 ($3.41 \times 10^{-2}$) | 0.96 ($3.59 \times 10^{-2}$) |
| ORT.js-WebGL-I | 10.5% ($0.39 \times 10^{-2}$) | 11.1% ($0.45 \times 10^{-2}$) | 7.9% ($0.30 \times 10^{-2}$) | / | / | / |
| ORT.js-WebGL-D | 15.1% ($0.57 \times 10^{-2}$) | 15.1% ($0.56 \times 10^{-2}$) | 13.5% ($0.54 \times 10^{-2}$) | / | / | / |

Each entry is the average classification accuracy or detection mAP across devices. The value in the bracket is the standard deviation of the metric. "M*," "/," and "-D/-I" indicate the same as Table 4. We found that in-browser inference can reduce classification accuracy by up to 9.1% and lower the detection mAP by up to 24%.

where humans are the predominant subject. We chose these two benchmark datasets specifically for their inclusion of both video frames sourced from real-world video streams and corresponding timestamps, and they are used in numerous research papers [61, 73, 91, 96, 102, 111].

When performing inference in the browser, we observe that the classification accuracy drops by up to 9.1%, and the detection mAP drops by up to 24%. Specifically, on CPUs, the reduction in classification accuracy and mAP for object detection reached up to 9.2% and 0.23, respectively. On integrated GPUs, the two metrics decreased by up to 13.3% and 0.24, respectively. On discrete GPUs, the reductions were up to 6.1% and 0.06, respectively. We found that these results are very similar to previous findings. Firstly, discrete GPUs consistently show the best results across all settings due to their performance advantages. In contrast, integrated GPUs perform worse than CPUs in terms of inference accuracy QoE. The main reason is that we considered the continuous inference scenario in the experiment, such as processing a video segment. If the inference is performed on integrated GPUs, it competes for resources with rendering tasks, leading to reduced rendering fps. This can force some video frames to reuse results from previous frames, deteriorating the quality of the output [62, 66, 83]. Therefore, for tasks that require frequent inference, such as background blur in online video meetings, using integrated GPUs might not be a better choice than CPU in terms of inference accuracy QoE. Although model inference on CPU is slower than that on integrated GPUs, the lesser competition for resources can achieve higher QoE through parallel processing.

Additionally, we compared the inference accuracy QoE of different models within the same task. In the image classification task, the MobileNetV2 model had the best inference accuracy, averaging

Table 21. QoE Measurement in Integrated Results

| | Baseline | TF.js-Wasm | ORT.js-Wasm | TF.js-WebGL-I | TF.js-WebGL-D | ORT.js-WebGL-I | ORT.js-WebGL-I |
|---|---|---|---|---|---|---|---|
| Responsiveness | 142.0 (2.3) | 104.2 (7.2) | 101.7 (10.1) | 94.2 (14.1) | 115.8 (11.5) | / | / |
| Smoothness | 30.0 (0.00) | 20.9 (1.82) | 21.4 (1.73) | 10.3 (2.84) | 23.8 (1.94) | / | / |
| Infer. Acc. | 1.0 (0.00) | 0.92 $(2.80 \times 10^{-2})$ | 0.91 $(3.10 \times 10^{-2})$ | 0.84 $(4.60 \times 10^{-2})$ | 0.98 $(4.60 \times 10^{-2})$ | / | / |

The format of each entry is the same as the previous tables. We found that with in-browser inference, all three QoE metrics decreased, with a greater decline observed when evaluated individually. Specifically, the responsiveness score dropped by 18.8 more, and the fps decreased by 4.5 more.

12.8%, while the VGG16 model performed the worst, averaging 8.9%. We examined how each model processed every video frame and found that the VGG16 model, due to its high prediction latency, led to more video frames having to reuse previous results, thereby degrading its performance. Conversely, the MobileNetV2 model reused the fewest results. Although the ResNet50 model has higher accuracy than MobileNetV2 on the ImageNet dataset, its performance deteriorated due to excessive frame reuse. In the object detection task, we found that the Yolo5-Middle model achieved the best inference accuracy, averaging 0.84, while the SSD-MobileNetV2 model showed the poorest performance, averaging 0.82. This is because SSD-MobileNetV2 is the only supported object detection model in the Wasm backend of TF.js, and the results on the CPU are poorer than the average. However, with respect to the other backends, EfficientDet has the worst performance. Since the detection tasks in our dataset were relatively straightforward, the performance of the EfficientDet model was only marginally lower than other models by at most 0.02. The Yolo5-Middle model, benefiting from its lower latency and superior pretrained accuracy, performed best in the inference accuracy QoE experiments. Therefore, to achieve higher inference accuracy QoE, it should be avoided to use heavier models. Lightweight models can achieve comparable outcomes and are preferable due to their efficiency and lesser resource demands.

In summary, in-browser inference leads to inference accuracy degradation. The constrained computational resources and reduced fps pose challenges for the DL model to process all inputs effectively. Performing inference in a slower backend may bring better inference accuracy.

*Finding 5*: In-browser inference can degrade the inference accuracy. We observe a decrease of up to 9.1% in classification accuracy and 24% in detection mAP. When performing inference in Wasm, applications could achieve higher inference accuracy at the cost of a minor latency increase, compared with inference in WebGL on integrated GPU. The results show that reducing inference latency does not necessarily equate to an improvement in inference accuracy QoE, especially for video applications that require continuous DL inference.

## 5.4 Analysis in an Integrated Environment

We conducted an analysis of three QoE metrics within a unified environment. Since we could not find any open source Web sites deployed with DL inference, we developed an online video conferencing Web application. This app captures video from the device's camera and displays it on the Web page. At the same time, we embedded the Speedometer benchmark test within the page. The Web page uses SSD-MobileNetV2 for face detection, monitoring the video's fps data in real time while detecting faces. The logic for online face detection remains the same as before, reusing previous results when the current frame cannot be processed. Additionally, we captured the entire monitor's display and used YoloV8 for detection to serve as ground truth. The results of this part of the experiment are displayed in Table 21.

We found that compared to previous standalone tests, the QoE results obtained from this part of the test were worse. First, regarding responsiveness, compared to the baseline (without in-browser inference), the average QoE score for responsiveness in this scenario decreased by up to 47.8, with the standard deviation increasing by up to 11.8, and compared to just deploying in-browser inference with the Speedometer benchmark test, the average QoE score dropped by up to 18.8. This is mainly because the benchmark test also included many rendering tasks, which also compete for resources with the benchmark test, hence worsening the QoE results. This demonstrates that resource competition can impact responsiveness. The difference between the two frameworks is very low; with TF.js as the baseline, the gap between ORT.js and TF.js is less than 2.4%. We also found that responsiveness scores were the lowest on integrated GPUs, primarily due to limited GPU performance and resource competition. As for the results for smoothness, compared to results without in-browser inference, fps in this scenario dropped by up to 19.7, and the standard deviation increased by up to 2.84. Compared to the scenario in Section 5.2, the fps in this scenario was lower, with the average fps being lower by up to 4.5. The reason remains the same as before, due to the impact of resource competition, as the Speedometer benchmark test also occupies CPU and GPU resources, thus worsening fps. As concluded in Section 5.2, integrated GPUs, being less powerful, showed the worst results. As for the results for inference accuracy, compared to the ideal case, where the DL model could process all video frames, the inference accuracy QoE, i.e., detection mAP, in this scenario decreased by up to 0.16. Since the dataset used differs from Section 5.3, a direct comparison is not possible. Nevertheless, our previous conclusion still holds that resource competition causes inference to reuse previous results, leading to impaired inference accuracy QoE.

In summary, after testing in a unified environment, we found that our previous conclusions hold true: the impact of resource competition can cause a decline in QoE. We can also see that each QoE metric declines when in-browser inference leads to resource competition. Therefore, ensuring QoE is a tradeoff and should be chosen based on specific application requirements. For video applications, smoothness usually has a higher priority, while for applications with intensive interactions, responsiveness is more important. Web application developers should adjust in-browser inference settings according to different application needs, such as choosing the backend and model to use.

> *Finding 6*: In-browser inference introduces inevitable resource competition, resulting in a decline across three QoE metrics. Compared to the baseline, in a unified environment, we observed that responsiveness QoE dropped by up to 47.8, smoothness QoE by up to 19.7, and inference accuracy by up to 0.16. Balancing these three QoE metrics involves tradeoffs and should be tailored based on the specific needs of the application, such as prioritizing page rendering smoothness. For applications where smoothness is a priority, using Wasm as the inference backend typically yields better results than integrated GPUs.

## 6 Implications and Suggestions

This section discusses a series of actionable implications based on our findings.

*For Web Browser Vendors: (1) Extend Wasm SIMD instructions (Finding 1).* Limited SIMD instructions in Wasm hinder the potential optimization. Browser vendors can consider extending the SIMD instructions with more features, such as FMA and data-moving instructions. This enhancement can empower the in-browser inference to leverage advanced hardware capabilities better. There are two possible approaches to extending SIMD capabilities for enhanced in-browser inference performance. The first is updating the Wasm SIMD specification to include longer SIMD instructions, aiming

for transparency in higher-level applications. This approach would require effort support from browser vendors to add support in browser Wasm VM and may also require effort within the Web community to refine the Wasm specification. The second is adding browser extensions for DL inference. As in-browser inference grows and more Web services integrate DL models, this approach requires to development of browser extensions specifically for DL models. These extensions could enable the browser itself to execute kernels like matrix multiplication directly, providing interfaces to the JavaScript side. Since browsers can directly access native resources, they are well positioned to implement extended SIMD capabilities. This method would only require support from browser vendors. (2) *Support loading precompiled WebGL binaries (Finding 1)*. Currently, the compilation of WebGL in browsers involves translating WebGL shader code into an intermediate representation, such as how Chrome on Windows platforms compiles WebGL into Direct3D [1], which is then compiled by the GPU hardware into executable binaries. This entire process occurs within the browser. However, the form of the intermediate representation is limited, and converting WebGL to intermediate form is time-consuming. Therefore, browser vendors could consider implementing a feature that allows the loading of precompiled WebGL intermediate representations directly into the browser. This would help avoid the time-consuming in-browser compilation of WebGL shaders, potentially reducing the warmup latency of in-browser inference. This improvement can not only benefit the in-browser inference but also enhance the performance of other Web applications.

*For DL Framework Vendors:* (1) *Compile WebGL binaries ahead-of-time (Finding 1)*. In-browser compilation introduces significant warmup overhead to in-browser inference. In browsers, WebGL is first compiled into an intermediate representation. For example, ANGLE can compile WebGL to Direct3D on Windows and Metal on OS X [1, 17, 25]. These intermediate representations are then converted into executable machine code by the hardware. Given that the number of these intermediate representations is relatively limited, framework developers can precompile WebGL programs into these representations. During inference, the framework can download and load the necessary representations into the browser. This approach helps save on compilation time and can alleviate the issue of high warmup latency. This suggestion complements and synergizes with the second suggestion for Web browser vendors. (2) *Focus more on memory-intensive kernels (Finding 1)*. Memory-intensive kernels can introduce high latency, underscoring the importance of paying attention to these kernels. For example, framework vendors can make efforts to fully utilize multithreading in the implementation of these kernels. (3) *Manage memory through an efficient memory pool (Finding 2)*. When the size of intermediate data surpasses the allocated memory block size, frameworks need to require more memory. To mitigate the problem, memory pool techniques, optimized by profiling inference process or DL model, can be employed for efficient memory management [113, 123]. Currently, the models used for in-browser inference require format conversion from native models. During the conversion, it is possible to gather information about the model's memory access patterns, including the kernels themselves and the amount of memory each kernel needs to utilize. This information can be saved and downloaded alongside the model conversion. During the setup stage, the framework can preallocate memory while loading the model, which can help reduce warmup latency. Since preallocation of memory and data transfer can occur concurrently, this process does not generate additional overhead. When allocating memory, the method pool proposed by Melon [113] can be utilized. This approach involves preallocating a large enough memory block and then placing data during the inference process at different locations within this block, as long as data reads and writes do not conflict. In this way, data can be accommodated within the block, avoiding redundant memory allocation and memory fragments as much as possible. Implementing this strategy could significantly optimize memory management, thereby enhancing the efficiency and performance of in-browser inference.

*For Web Application Developers:* (1) *Incorporate DL models with varying capacities (Finding 1).* Web application developers can encapsulate multiple models within the application, each with different sizes and capacities, such as MobileNet [79, 101] and ResNet [78] catering to image classification, and download the suitable model according to hardware information. Specifically, on the GPU side, developers can use the "WEBGL_debug_renderer_info" extension [4] to obtain information about the GPU, and then select the suitable model based on this information. For instance, if a discrete GPU is detected, a model with powerful capacity can be chosen, whereas a lighter model may be preferable for integrated GPUs. On the CPU side, since it is difficult for developers to directly obtain specifications such as the CPU model, a performance benchmark can be constructed in advance. This benchmark involves a fixed-size matrix multiplication kernel (or other DL kernels) and measures its latency on CPUs of varying capacities. By executing this kernel during the Web page loading process and estimating the performance of the user's CPU based on the kernel's latency, it becomes possible to choose either a lightweight model or a model with a more powerful capacity according to the assessed capability of the CPU. (2) *Prioritize responsiveness (Finding 3).* Due to the resource contention generated by handling user requests and DL model inference, adopting intermittent inference can alleviate resource competition, ensuring the timely responsiveness of Web pages to user requests. *(3) Enhance smooth rendering (Finding 4).* Since rendering is typically GPU-dependent, using the Wasm backend for inference could guarantee smooth rendering. While Wasm has slightly higher inference latency than WebGL, it circumvents rendering resource competition, especially for devices with only integrated GPUs. This can introduce a significant improvement in Web page smoothness at the cost of a little inference latency. (4) *Guarantee inference accuracy (Finding 5).* For the video applications we used, frame filtering techniques [86] can effectively enhance inference accuracy. Additionally, utilizing Wasm as the inference backend is a practical choice, aligning with the previous suggestions and the findings. (5) *Balancing resource allocation for optimal QoE (Finding 6).* It is important to note that, deploying DL inference in browsers inevitably leads to resource competition, with inference tasks occupying either the CPU or GPU. This necessitates the consideration of inference configurations based on the specific applications involved. For example, while rendering tasks rely heavily on the GPU, user interactions depend more on the CPU. Therefore, ensuring optimal QoE across all three metrics becomes a tradeoff. Prioritizing GPU for rendering will consume more CPU resources, and *vice versa*. Developers can choose the appropriate QoE metric based on the application type, such as video playback or online document. Then developers can select the optimal backend to optimize the QoE metric based on the specific QoE metric. Responsiveness QoE can be quantified through the response time to user request, which is equal to the "runs per minute" defined in the article; smoothness QoE can be quantified through fps, which can be obtained by the JavaScript API "requestAnimationFrame." As for inference accuracy QoE, developers can quantify it through user feedback because there may not be ground truth for user inputs. For instance, if a developer is developing Web pages for video playback, the smoothness QoE should be prioritized, meaning that ensuring smooth video rendering should come first. When incorporating in-browser inference, it is essential to maximize inference accuracy QoE while maintaining smooth rendering. In such scenarios, the importance of responsiveness QoE may not be as significant as the other two. At this point, choosing CPU as the inference backend may be given higher priority. Developers can also optimize the QoE by monitoring the fps of video playback to select either a lightweight or more capable model. When developing Web pages for online documents, responsiveness QoE should be the primary focus, ensuring that timely response to user interactions. When introducing in-browser inference, it is necessary to maximize inference accuracy QoE without compromising responsiveness QoE, where smoothness QoE may become less critical compared to responsiveness. In this scenario, GPU could usually be a better choice.

Depending on the type of Web page, developers can prioritize different QoE metrics based on user requirements and feedback.

## 7 Discussion

### 7.1 Threats to Validity

*Selection of Models.* Similar to previous work [92], we select models that cover several classical tasks, and the models are widely deployed to user devices. We excluded LSTM, speech-to-text, and GPT models because TF.js and ORT.js do not support them. In addition, Bert outperforms LSTM in many language tasks [70]. Because models can be represented as a sequence of kernels [29, 56] and the basic kernel set is the same, we believe such selection introduces negligible bias in our results with respect to inference performance.

*Model Format Conversion.* We used the models from the TF model hub for different tasks. These models were pretrained on servers using TF and then converted to the file format recognizable by the in-browser inference framework using model conversion tools. Although TF.js supports training, its efficiency of model training on servers is significantly lower than TF, and the computational costs of model training in browsers make it impractical. The conversion process does not modify the model architecture or its kernels, and thus it does not impact inference performance. The models in the model hub are not specifically designed for in-browser inference, which may lead to conversion failures primarily due to model design. In practical deployments, developers can design models based on the specific needs of the applications. Although TF.js and ORT.js might not support certain models, we believe this will not hinder the progress of in-browser inference. Furthermore, our experimental results are independent of the model format conversion, affirming the validity of our results and findings.

*Selection of Frameworks.* We select the two frameworks based on multiple considerations, including documentation, popularity, backend support, and maintenance. For example, Brain.js [11] lacks support for CPU; Keras.js is no longer maintained; WebDNN [48] lacks implementation of most kernels in TF and ORT. TF.js is compatible with original TF models, whereas ORT.js handles models in the ONNX format. Popular DL frameworks, including PyTorch and TensorFlow, allow for their models to be converted to ONNX, a necessary step for mobile deployment in the case of PyTorch. As such, TF.js and ORT.js are capable of supporting most in-browser inference applications. These frameworks are robustly maintained over the long term due to their development under the auspices of Google and Microsoft, respectively. In contrast, the slower update pace of other frameworks can be attributed to the lack of dedicated maintenance teams and the high costs associated with keeping up with rapid model iterations. Because of the extensive adoption of in-browser inference and strong community support for TF.js and ORT.js, we believe that in-browser inference is promising. Although we exclude ORT.js in the kernel-level latency and memory footprint analysis, the underlying logic of the DL framework is similar, i.e., computing each kernel of the model. The kernel computation logic is the same and the latency distribution is framework-agnostic. Based on these points, we believe our results are valid.

*Selection of Tasks.* We select vision and text tasks for the experiment. These tasks are popular on the Web and there have been similar deployed services. Despite the "inference accuracy" in the QoE experiment being task-specific, the inference accuracy will degrade inevitably due to resource competition. We involve super-resolution tasks only in the memory footprint analysis because of the immature support. However, excluding this task in other parts does not introduce too much bias to the final conclusions, i.e., large latency gaps compared with native inference, large latency variations across devices, and degradation of the user QoE. So we believe our results and findings are still reliable. Regarding the experimental setup, our focus was on the performance

of model inference, which is independent of application logic. Deployed real-world applications are not open-sourced because of the confidentiality requirements of commercial applications. Nevertheless, considering that model inference performance is influenced solely by hardware and browser capabilities, we designed Web pages specifically for measuring inference performance, containing only the logic related to model inference.

*Selection of Devices.* We used 50 PC devices and 20 mobile devices for the experiment. The CPU and GPU of the devices cover 82% devices in AI Benchmark [9] w.r.t. the AI score. Although the device selection may introduce bias to results, we believe the bias is acceptable and the results and findings are still valid.

## 7.2 Future Work

Optimizing in-browser inference performance is a crucial issue. Our analysis has already indicated that on the CPU side, the main challenges during the prediction stage of in-browser inference stem from environmental differences between native and browser settings. Future work could focus on addressing these issues. For instance, expanding the SIMD instruction set in Wasm to support more efficient operations, such as FMA, could enhance performance significantly. Another potential approach could be to bypass Wasm altogether and implement model kernels, such as matrix multiplication, directly within the browser's JavaScript engine. Since browsers can directly access advanced hardware features and bypass the Wasm interpreter, this could substantially improve inference performance. We are currently exploring optimizations in this direction. For the warmup stage, considering the high in-browser compilation overhead associated with WebGL, introducing support for importing precompiled intermediate representations into the browser could significantly reduce latency during this stage. This approach would streamline the process, allowing for quicker start-up times and enhancing the overall responsiveness of Web applications utilizing in-browser inference. We leave this as a future direction to further optimize the in-browser inference performance. As for the memory footprint of in-browser inference, we aim to optimize the memory management mechanism of the inference framework. Specifically, this optimization mainly includes the following steps: (1) First, during the model conversion phase, it is necessary to retain the model structure and information about each kernel. (2) Based on this information, obtain the memory access patterns and use optimized memory pool techniques to precalculate the strategy of memory management and allocation [113]. (3) Save this information along with the model files so that the inference framework can download and reduce the inference memory footprint.

In summary, our future work primarily focuses on optimizing the performance of in-browser inference, as enhancing inference performance will inevitably lead to improvements in QoE. For instance, lower latency means that devices have more time to process user requests and the DL model can handle more video frames. Additionally, conserving memory footprint will allow browsers to operate more smoothly, avoid stuttering, and thus enhance the smoothness QoE.

## 8 Related Work

*DL Performance.* DL performance is critical to DL-powered software development and deployment, as well as to end users' QoE. For example, high accuracy and low inference latency are desired [60, 116]. To characterize the DL performance, many research efforts in the SE community have been presented [63, 64, 72, 74, 76, 97]. For example, DeepPerf [76] is an end-to-end DL-based solution that can predict the performance of a new DL task through a pretrained model; DeepPerform [64] enables efficient performance testing for AdNN [58]; DNNMem [74] estimates the memory consumption of both the DL model and the DL framework systematically and accurately. Additionally, Ma et al. [92] measured the in-browser inference latency at model level for manually constructed

models. Currently, most research focuses on inference performance in native environments, with a lack of detailed analysis of inference performance within browser environments. Unlike native environments, browsers isolate the inference process from the operating system, preventing access to advanced system features, such as efficient SIMD instructions. Our research aims to fill this knowledge gap by exploring inference performance from the model level to the kernel level, covering aspects such as latency and memory footprint. While Ma et al. [92] explored manually constructed models, they neither considered models and applications deployed in the real world nor analyzed and compared the differences between browser and native environments. In contrast, our study is designed to thoroughly dissect in-browser inference of real-world DL models at both the model and kernel levels, as well as explore the memory footprint and user QoE. This comprehensive approach allows us to provide deeper insights and recommendations for optimizing in-browser inference performance.

*DL Deployment.* DL deployment focuses on fitting models into the different platforms after the DL model is well tested [65, 75, 94, 116]. Many frameworks facilitate the deployment in the cloud, such as TensorFlow [56] for servers, TFLite [40] for Android, and Core ML [14] for iOS. Many SE efforts are proposed to promote DL deployment [67–69, 75]. For example, Cummaudo et al. [69] analyzed pain points that developers face when deploying DL to servers. Chen et al. [67, 68] studied the DL deployment issues and challenges by exploring forums such as Stack Overflow. Guo et al. [75] investigated the performance disparity when the pretrained DL models are migrated to mobile and Web platforms. They analyze the model-level performance in the pure JavaScript backend. Most existing research still focuses on native environments. Although Guo et al. [75] considered DL model deployment using pure JavaScript backend, they did not explore advanced backends such as Wasm. At present, advanced backends are widely adopted for in-browser inference due to their superior performance capabilities. However, to our best knowledge, these backends have not yet been thoroughly investigated. In contrast, our study aims to delve into the performance of more advanced backends, i.e., Wasm and WebGL. We seek to understand the reasons behind the performance disparities between browser and native environments, provide detailed analysis at the fine-grained kernel level, and assess the impact of in-browser inference on user QoE.

*Web Browsing Performance.* Many metrics have been proposed to quantify the Web browsing performance, such as PLT [28], Time-to-First-Byte [42], Speed Index [36], and so on. Several efforts have been presented to explore and optimize the performance. Lighthouse [24] is a tool to analyze the quality of Web pages and is integrated into Chrome. WProf [115] is proposed to profile the PLT, which is a key performance metric that many techniques aim to reduce [85, 93]. SipLoader [90] is a Speed Index oriented page scheduler. Bocchi et al. proposed a comprehensive study to measure the QoE of Web users [59]. Vetter [118] is an automated tool for visual distortions testing and debugging. Current research focusing on browser performance and QoE primarily targets the Web page loading phase or concerns visual distortions. None of these studies have considered scenarios involving the deployment of DL inference. The loading phase occurs only at the beginning of Web page navigation, whereas, by comparison, DL inference continues throughout the user's interaction with the page. Existing studies have not considered this scenario, and the existing QoE metrics are not applicable. In contrast, our work is the first to propose QoE metrics tailored for in-browser scenarios and carries out a detailed quantitative analysis of these metrics.

## 9 Conclusion

We anatomized the in-browser inference performance and its impact on QoE. Results show that (1) the latency gap between in-browser inference and native inference reaches up to 16.9× on CPU and 30.6× on GPU. The discrepancy in CPU is mainly attributed to different SIMD instruction sets and Wasm runtime-inherent inefficiency, as well as resource competition within the browser; the

discrepancy in GPU is mainly attributed to inefficient software libraries and GPU abstractions. (2) Memory can be the bottleneck. When inferring super-resolution models, the memory requirement can reach up to 334.6× the size of the models themselves. (3) QoE will degrade due to inevitable resource competition. Inferring models on CPU may bring better QoE due to the tradeoff between inference latency and QoE.

## References

[1] ANGLE (software). 2024. Retrieved April 2024 from https://en.wikipedia.org/wiki/ANGLE_(software)
[2] Selenium with Python. 2024. Retrieved April 2024 from https://selenium-python.readthedocs.io/
[3] Using the SavedModel Format. 2024. Retrieved April 2024 from https://www.tensorflow.org/guide/saved_model
[4] WEBGL_debug_renderer_info Extension. 2024. Retrieved April 2024 from https://developer.mozilla.org/en-US/docs/Web/API/WEBGL_debug_renderer_info
[5] XNNPACK: High-Efficiency Floating-Point Neural Network Inference Operators for Mobile, Server, and Web. 2024. Retrieved April 2024 from https://github.com/google/XNNPACK
[6] Grammar Checking. 2024. Retrieved July 2024 from https://www.microsoft.com/en-us/microsoft-365/microsoft-editor/grammar-checker
[7] Teams Online. 2024. Retrieved July 2024 from https://teams.microsoft.com/
[8] Theremix. 2024. Retrieved July 2024 from https://theremin.app/
[9] AI Benchmark for Windows, Linux and macOS. 2023. Retrieved September 2023 from https://ai-benchmark.com/ranking_deeplearning_detailed.html
[10] AI Face Test. 2023. Retrieved September 2023 from https://facetest1.netlify.app/
[11] Brain.js: GPU Accelerated Neural networks in JavaScript for Browsers and Node.js. 2023. Retrieved September 2023 from https://brain.js.org/
[12] Browser Market Share Worldwide. 2023. Retrieved September 2023 from https://gs.statcounter.com/browser-market-share
[13] Change Your Background in Microsoft Teams Meetings. 2023. Retrieved September 2023 from https://support.microsoft.com/en-us/office/change-your-background-in-microsoft-teams-meetings-f77a2381-443a-499d-825e-509a140f4780
[14] Core ML. 2023. Retrieved September 2023 from https://developer.apple.com/documentation/coreml
[15] CPU Benchmarks – Over 1,000,000 CPUs benchmarked. 2023. Retrieved September 2023 from https://www.cpubenchmark.net/
[16] CUDA. 2023. Retrieved September 2023 from https://en.wikipedia.org/wiki/CUDA
[17] Direct3D. 2023. Retrieved September 2023 from https://en.wikipedia.org/wiki/Direct3D
[18] Face Age Test. 2023. Retrieved September 2023 from https://aige.netlify.app/
[19] FMA Instruction Set. 2023. Retrieved September 2023 from https://en.wikipedia.org/wiki/FMA_instruction_set
[20] Getting Started Converting TensorFlow to ONNX. 2023. Retrieved September 2023 from https://onnxruntime.ai/docs/tutorials/tf-get-started.html
[21] How WebAssembly Threads Work. 2023. Retrieved September 2023 from https://web.dev/webassembly-threads/
[22] Intel AVX-512 Instructions. 2023. Retrieved September 2023 from https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html
[23] Intel AVX2 Instructions. 2023. Retrieved September 2023 from https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx2.html
[24] Lighthouse. 2023. Retrieved September 2023 from https://developer.chrome.com/docs/lighthouse/overview/
[25] Metal: Accelerating Graphics and Much More. 2023. Retrieved September 2023 from https://developer.apple.com/metal/
[26] Model Conversion. 2023. Retrieved September 2023 from https://www.tensorflow.org/js/guide/conversion
[27] Module: tf.keras.applications. 2023. Retrieved September 2023 from https://www.tensorflow.org/api_docs/python/tf/keras/applications
[28] Navigation Timing API. 2023. Retrieved September 2023 from https://developer.mozilla.org/en-US/docs/Web/API/Navigation_timing_API
[29] ONNX. 2023. Retrieved September 2023 from https://onnx.ai/
[30] ONNX Runtime: Optimize and Accelerate Machine Learning Inferencing and Training. 2023. Retrieved September 2023 from https://onnxruntime.ai/
[31] ONNX Web Runtime. 2023. Retrieved September 2023 from https://onnxruntime.ai/

[32] OpenGL: The Industry's Foundation for High Performance Graphics. 2023. Retrieved September 2023 from https://www.opengl.org/

[33] Relaxed SIMD Proposal. 2023. Retrieved September 2023 from https://github.com/WebAssembly/relaxed-simd/blob/main/proposals/relaxed-simd/Overview.md

[34] Rocket Lake. 2023. Retrieved September 2023 from https://en.wikipedia.org/wiki/Rocket_Lake

[35] Skylake (Microarchitecture). 2023. Retrieved September 2023 from https://en.wikipedia.org/wiki/Skylake_(microarchitecture)

[36] Speed Index. 2023. Retrieved September 2023 from https://developer.chrome.com/docs/lighthouse/performance/speed-index/

[37] Speedometer. 2023. Retrieved September 2023 from https://browserbench.org/Speedometer2.1/

[38] Technical City: Unbiased Hardware Comparisons. 2023. Retrieved September 2023 from https://technical.city/

[39] TensorFlow Hub. 2023. Retrieved September 2023 from https://tfhub.dev/

[40] TensorFlow Lite. 2023. Retrieved September 2023 from https://www.tensorflow.org/lite

[41] TensorFlow.js. 2023. Retrieved September 2023 from https://www.tensorflow.org/js

[42] Time to First Byte. 2023. Retrieved September 2023 from https://developer.mozilla.org/en-US/docs/Glossary/time_to_first_byte

[43] Using Web Workers. 2023. Retrieved September 2023 from https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

[44] V8. 2023. Retrieved September 2023 from https://v8.dev/

[45] Web LLM. 2023. Retrieved September 2023 from https://github.com/mlc-ai/web-llm

[46] WebAssembly. 2023. Retrieved September 2023 from https://webassembly.org/

[47] WebAssembly 128-Bit Packed SIMD Extension. 2023. Retrieved September 2023 from https://github.com/WebAssembly/spec/blob/main/proposals/simd/SIMD.md

[48] WebDNN. 2023. Retrieved September 2023 from https://mil-tokyo.github.io/webdnn/

[49] WebGL: 2D and 3D Graphics for the Web. 2023. Retrieved September 2023 from https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

[50] WebGL2RenderingContext. 2023. Retrieved September 2023 from https://developer.mozilla.org/en-US/docs/Web/API/WebGL2RenderingContext

[51] WebGL2RenderingContext: getSyncParameter() method. 2023. Retrieved September 2023 from https://developer.mozilla.org/en-US/docs/Web/API/WebGL2RenderingContext/getSyncParameter

[52] WebGLShader. 2023. Retrieved September 2023 from https://developer.mozilla.org/en-US/docs/Web/API/WebGLShader

[53] WebGPU-W3C Working Draft. 2023. Retrieved September 2023 from https://www.w3.org/TR/webgpu/

[54] YouTube-BoundingBoxes Dataset. 2023. Retrieved September 2023 from https://research.google.com/youtube-bb/

[55] YouTube Faces DB. 2023. Retrieved September 2023 from https://www.cs.tau.ac.il/wolf/ytfaces/

[56] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 265–283.

[57] Amin Ahmad, Noah Constant, Yinfei Yang, and Daniel Cer. 2019. ReQA: An evaluation for end-to-end answer retrieval models. arXiv:1907.04780. Retrieved from https://doi.org/10.48550/arXiv.1907.04780

[58] Soroush Bateni and Cong Liu. 2018. ApNet: Approximation-aware real-time neural network. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 67–79.

[59] Enrico Bocchi, Luca De Cicco, and Dario Rossi. 2016. Measuring the quality of experience of web users. *ACM SIGCOMM Computer Communication Review* 46, 4 (2016), 8–13.

[60] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. 2019. Towards federated learning at scale: System design. *Proceedings of Machine Learning and Systems* 1 (2019), 374–388.

[61] Qi Cai, Yingwei Pan, Chong-Wah Ngo, Xinmei Tian, Lingyu Duan, and Ting Yao. 2019. Exploring object relation in mean teacher for cross-domain detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 11457–11466.

[62] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G Andersen, Michael Kaminsky, and Subramanya Dulloor. 2019. Scaling video analytics on constrained edge nodes. *Proceedings of Machine Learning and Systems* 1 (2019), 406–417.

[63] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding performance problems in deep learning systems. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 357–369.

[64] Simin Chen, Mirazul Haque, Cong Liu, and Wei Yang. 2022. DeepPerform: An efficient approach for performance testing of resource-constrained neural networks. In *37th IEEE/ACM International Conference on Automated Software Engineering*, 1–13.

[65] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, 578–594.

[66] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. 2015. Glimpse: Continuous, real-time object recognition on mobile devices. In *13th ACM Conference on Embedded Networked Sensor Systems*, 155–168.

[67] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A comprehensive study on challenges in deploying deep learning based software. In *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 750–762.

[68] Zhenpeng Chen, Huihan Yao, Yiling Lou, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, and Xuanzhe Liu. 2021. An empirical study on deployment faults of deep learning based mobile applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 674–685.

[69] Alex Cummaudo, Rajesh Vasa, Scott Barnett, John Grundy, and Mohamed Abdelrazek. 2020. Interpreting cloud computer vision pain-points: A mining study of stack overflow. In *ACM/IEEE 42nd International Conference on Software Engineering*, 1584–1596.

[70] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805. Retrieved from https://doi.org/10.48550/arXiv.1810.04805

[71] Bing Dong, Tianen Liu, Borui Li, Xiaolei Zhou, Shuai Wang, and Zhao-Dong Xu. 2023. WebInf: Accelerating WebGPU-based in-browser DNN inference via adaptive model partitioning. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2499–2506.

[72] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. DeepStellar: Model-based quantitative analysis of stateful deep learning systems. In *2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 477–487.

[73] Heng Fan, Liting Lin, Fan Yang, Peng Chu, Ge Deng, Sijia Yu, Hexin Bai, Yong Xu, Chunyuan Liao, and Haibin Ling. 2019. Lasot: A high-quality benchmark for large-scale single object tracking. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 5374–5383.

[74] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. 2020. Estimating GPU memory consumption of deep learning models. In *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1342–1352.

[75] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 810–822.

[76] Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance prediction for configurable software with deep sparse neural network. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1095–1106.

[77] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 185–200.

[78] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 770–778.

[79] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. 2019. Searching for MobileNetV3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 1314–1324.

[80] Yakun Huang, Xiuquan Qiao, Jian Tang, Pei Ren, Ling Liu, Calton Pu, and Jun-Liang Chen. 2021. An integrated cloud-edge-device adaptive deep learning service for cross-platform web. *IEEE Transactions on Mobile Computing* 22, 4 (2021), 1950–1967.

[81] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. 2019. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*, 107–120.

[82] Fucheng Jia, Shiqi Jiang, Ting Cao, Wei Cui, Tianrui Xia, Xu Cao, Yuanchun Li, Qipeng Wang, Deyu Zhang, Ju Ren, Yunxin Liu, Lili Qiu, and Mao Yang. 2024. Empowering in-browser deep learning inference on Edge through

Just-In-Time kernel optimization. In *22nd Annual International Conference on Mobile Systems, Applications and Services*, 438–450.

[83] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing neural network queries over video at scale. arXiv:1703.02529. Retrieved from https://doi.org/10.48550/arXiv.1703.02529

[84] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2017. Progressive growing of GANs for improved quality, stability, and variation. arXiv:1710.10196. Retrieved from https://doi.org/10.48550/arXiv.1710.10196

[85] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R. Das. 2017. Improving user perceived page load times using gaze. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, 545–559.

[86] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. 2020. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, 359–376.

[87] Chen Ling, Lei Wang, Jun Lang, Qiufen Xia, Guoxuan Chang, Kun Wang, and Peng Zhao. 2018. LinCa: A page loading time optimization approach for users subject to Internet access restriction. In *Companion Proceedings of the the Web Conference 2018*, 69–70.

[88] Ting Liu, Tianhao Miao, Qinghua Wu, Zhenyu Li, Guangxin He, Jiaoren Wu, Shengzhuo Zhang, Xingwu Yang, Gareth Tyson, and Gaogang Xie. 2022. Modeling and optimizing the scaling performance in distributed deep learning training. In *ACM Web Conference 2022*, 1764–1773.

[89] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single shot multibox detector. In *European Conference on Computer Vision*. Springer, 21–37.

[90] Wei Liu, Xinlei Yang, Hao Lin, Zhenhua Li, and Feng Qian. 2022. Fusing speed index during Web page loading. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–23.

[91] Xin Liu, Fatemeh Karimi Nejadasl, Jan C. van Gemert, Olaf Booij, and Silvia L. Pintea. 2023. Objects do not disappear: Video object detection by single-frame object location anticipation. In *IEEE/CVF International Conference on Computer Vision*, 6950–6961.

[92] Yun Ma, Dongwei Xiang, Shuyu Zheng, Deyu Tian, and Xuanzhe Liu. 2019. Moving deep learning into web browser: How far can we go? In *The World Wide Web Conference*, 1234–1244.

[93] Shaghayegh Mardani, Ayush Goel, Ronny Ko, Harsha V Madhyastha, and Ravi Netravali. 2021. Horcrux: Automatic JavaScript parallelism for resource-efficient web computation. In *15th USENIX Symposium on Operating Systems Design and Implementation OSDI '21*, 461–477.

[94] Gaurav Mittal, Kaushal B. Yagnik, Mohit Garg, and Narayanan C. Krishnan. 2016. Spotgarbage: Smartphone app to detect garbage using deep learning. In *2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 940–945.

[95] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2016. Pruning convolutional neural networks for resource efficient inference. arXiv:1611.06440. Retrieved from https://doi.org/10.48550/arXiv.1611.06440

[96] Tomas Pfister, James Charles, and Andrew Zisserman. 2015. Flowing convnets for human pose estimation in videos. In *IEEE International Conference on Computer Vision*, 1913–1921.

[97] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. Problems and opportunities in training deep learning software systems: An analysis of variance. In *35th IEEE/ACM International Conference on Automated Software Engineering*, 771–783.

[98] Ihsan Ayyub Qazi, Zafar Ayyub Qazi, Theophilus A. Benson, Ghulam Murtaza, Ehsan Latif, Abdul Manan, and Abrar Tariq. 2020. Mobile web browsing under memory pressure. *ACM SIGCOMM Computer Communication Review* 50, 4 (2020), 35–48.

[99] Lili Quan, Qianyu Guo, Xiaofei Xie, Sen Chen, Xiaohong Li, and Yang Liu. 2022. Towards understanding the faults of JavaScript-based deep learning systems. In *37th IEEE/ACM International Conference on Automated Software Engineering*, 1–13.

[100] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *IEEE Conference on Computer Vision and Pattern Recognition*, 779–788.

[101] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 4510–4520.

[102] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition*, 815–823.

[103] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556. Retrieved from https://doi.org/10.48550/arXiv.1409.1556

[104] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. MobileBert: A compact task-agnostic Bert for resource-limited devices. arXiv:2004.02984. Retrieved from https://doi.org/10.48550/arXiv.2004.02984

[105] Mingxing Tan, Ruoming Pang, and Quoc V. Le. 2020. EfficientDet: Scalable and efficient object detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 10781–10790.

[106] Xiaohu Tang, Yang Wang, Ting Cao, Li Lyna Zhang, Qi Chen, Deng Cai, Yunxin Liu, and Mao Yang. 2023. LUT-NN: Empower efficient neural network inference with centroid learning and table lookup. In *29th Annual International Conference on Mobile Computing and Networking*, 1–15.

[107] Deyu Tian and Yun Ma. 2019. Understanding quality of experiences on different mobile browsers. In *11th Asia-Pacific Symposium on Internetware*, 1–10.

[108] Deyu Tian, Haiyang Shen, and Yun Ma. 2022. Parallelizing DNN inference in mobile web browsers on heterogeneous hardware. In *20th Annual International Conference on Mobile Systems, Applications and Services*, 519–520.

[109] Bipin Upadhyaya, Ying Zou, Iman Keivanloo, and Joanna Ng. 2015. Quality of experience: User's perception about web services. *IEEE Transactions on Services Computing* 8, 3 (2015), 410–421.

[110] Krishna Wadhwani and Tamaki Kojima. 2022. SqueezeNeRF: Further factorized FastNeRF for memory-efficient inference. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2717–2725.

[111] Hao Wang, Yitong Wang, Zheng Zhou, Xing Ji, Dihong Gong, Jingchao Zhou, Zhifeng Li, and Wei Liu. 2018. CosFace: Large margin cosine loss for deep face recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 5265–5274.

[112] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. 2021. AsyMo: Scalable and efficient deep-learning inference on asymmetric mobile CPUs. In *27th Annual International Conference on Mobile Computing and Networking*, 215–228.

[113] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. 2022. Melon: Breaking the memory wall for resource-efficient on-device machine learning. In *20th Annual International Conference on Mobile Systems, Applications and Services*, 450–463.

[114] Xintao Wang, Ke Yu, Shixiang Wu, Jinjin Gu, Yihao Liu, Chao Dong, Yu Qiao, and Chen Change Loy. 2018. ESRGAN: Enhanced super-resolution generative adversarial networks. In *European Conference on Computer Vision (ECCV) Workshops*.

[115] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying page load performance with WProf. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, 473–485.

[116] Mengwei Xu, Feng Qian, Qiaozhu Mei, Kang Huang, and Xuanzhe Liu. 2018. DeepType: On-device deep learning for input personalization service with minimal privacy concern. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 4 (2018), 1–26.

[117] Mengwei Xu, Xiwen Zhang, Yunxin Liu, Gang Huang, Xuanzhe Liu, and Felix Xiaozhu Lin. 2020. Approximate query service on autonomous IOT cameras. In *18th International Conference on Mobile Systems, Applications, and Services*, 191–205.

[118] Xinlei Yang, Wei Liu, Hao Lin, Zhenhua Li, Feng Qian, Xianlong Wang, Yunhao Liu, and Tianyin Xu. 2023. Visual-aware testing and debugging for web performance optimization. In *ACM Web Conference 2023*, 2948–2959.

[119] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2022. A survey on deep learning for software engineering. *ACM Computing Surveys (CSUR)* 54, 10s (2022), 1–73.

[120] Jihwan Yeo, Changhyun Shin, and Soo-Mook Moon. 2019. Snapshot-based loading acceleration of web apps with nondeterministic JavaScript execution. In *The World Wide Web Conference*, 2215–2224.

[121] Qiyang Zhang, Xiang Li, Xiangying Che, Xiao Ma, Ao Zhou, Mengwei Xu, Shangguang Wang, Yun Ma, and Xuanzhe Liu. 2022. A comprehensive benchmark of deep learning libraries on mobile devices. In *ACM Web Conference 2022*, 3298–3307.

[122] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing deep learning applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 104–115.

[123] Quan Zhou, Haiquan Wang, Xiaoyan Yu, Cheng Li, Youhui Bai, Feng Yan, and Yinlong Xu. 2023. MPress: Democratizing billion-scale model training on multi-GPU servers via memory-saving inter-operator parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 556–569.

[124] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. 2017. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision*, 2223–2232.