A Comprehensive Benchmark of Deep Learning Libraries on Mobile Devices

Qiyang Zhang[1], Xiang Li[2], Xiangying Che[1], Xiao Ma[1], Ao Zhou[1], Mengwei Xu[1], Shangguang Wang[1],Yun Ma[3], Xuanzhe Liu[3]
Beijing University of Posts and Telecommunications[1],China
China University of Petroleum[2],China
Peking University[3],China

Abstract

Deploying deep learning (DL) on mobile devices has been a notable trend in recent years. To support fast inference of on-device DL, DL libraries play a critical role as algorithms and hardware do. Unfortunately, no prior work ever dives deep into the ecosystem of modern DL libs and provides quantitative results on their performance. In this paper, we first build a comprehensive benchmark that includes 6 representative DL libs and 15 diversified DL models. We then perform extensive experiments on 10 mobile devices, which help reveal a complete landscape of the current mobile DL libs ecosystem. For example, we find that the best-performing DL lib is severely fragmented across different models and hardware, and the gap between those DL libs can be rather huge. In fact, the impacts of DL libs can overwhelm the optimizations from algorithms or hardware, e.g., model quantization and GPU/DSP-based heterogeneous computing. Finally, atop the observations, we summarize practical implications to different roles in the DL lib ecosystem.

CCS Concepts

• **General and reference** → **Measurement**; • **Human-centered computing** → **Ubiquitous and mobile devices**.

Keywords

Benchmark, Deep Learning, Mobile Devices

ACM Reference Format:
Qiyang Zhang, Xiang Li, Xiangying Che, Ao Zhou, Mingwei Xu, Shangguang Wang, Yun Ma, Xuanzhe Liu. 2022. A Comprehensive Benchmark of Deep Learning Libraries on Mobile Devices. In *Proceedings of the ACM Web Conference 2022 (WWW '22), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3485447.3512148

1 Introduction

Deep learning has become a key enabler towards ubiquitous and intelligent web applications like Web AR [52, 54, 73, 78]. A noteworthy trend is that more and more Deep learning inference tasks are now shifting from cloud datacenters to smartphones, making a case for low user-perceived delay and data privacy preservation with the support of on-device DL [52]. For example, it is reported that the DL-embedded apps on Google Play market have increased by 60% from Feb. 2020 to Apr. 2021, and those apps contribute to billions of downloads and user reviews [22, 72].

Benchmark	Scenario	Benchmark objective	Supported DL libs
MLPerf [48]	T/I@S/E	Hardware	/
DeepBench [16]	T/I@S/E	Hardware	/
DAWNBench [15]	T/I@S	Hardware, algorithm, and DL libs	/
AI Matrix [7]	I@S	Hardware and DL libs	4
AI-Benchmark [34]	I@E	Hardware	1
Fathom [19]	T/I@S	Algorithm	1
gaugeNN [22]	I@E	DL apps and models	1
AIIA [13]	I@E	Hardware	3
This work	I@E	DL libs	6

Table 1: Comparison of this work and existing AI benchmarks. "T/I": training/inference stages; "S/E": server/edge platform. "/" means the benchmark ask the third-party users to submit their own software.

Running inference (or prediction) task in a fast way is the intuitively basic demand to on-device DL, as many of them are deployed for continuous user interactions. It is also fundamentally challenging because DL models are known to be very complex and cumbersome [21, 30, 51]. Consequently, optimizing the inference performance has been the theme of both academia [22, 73, 77] and industry [8, 12, 14] in recent years.

The inference performance of on-device DL is affected by many factors. Existing literature that aims to quantitatively understand the performance mostly focuses on hardware and models, leaving the software (DL execution engines or *DL libs*) underexplored. Yet, software also plays a critical role in speeding up the on-device DL inference, e.g., up to 62,806× gap between vanilla and fine-tuned implementation [42].

Furthermore, due to the severely fragmented ecosystem of smartphones [67], there exists a mass of heterogeneous DL lib alternatives for app developers [22, 72], making it difficult and labor-intensive to compare their suitability into specific models.

To gain in-depth understandings of the performance of modern DL libs, we first build a comprehensive benchmark for on-device DL inference, namely `MDLBench`. The benchmark includes 6 popular, representative DL libs on mobile devices, i.e., `TFLite`, `PyTorchMobile`, `ncnn`, `MNN`, `Mace`, and `SNPE` [1–6]. It contains 6 DL models compatible with all above DL libs and 8 models compatible with at least 3 above DL libs, spanning from image classification, object detection, to NLP. Compared to existing AI benchmarks (Table 1), our benchmark triumphs at the aspect of rich support for various DL libs and models. In addition to the completeness, we also

WWW '22, April 25–29, 2022, Virtual Event, Lyon, France.

ACM ISBN 978-1-4503-9096-5/22/04…$15.00
https://doi.org/10.1145/3485447.3512148

instrument the DL libs to obtain underlying performance details such as per-operator latency, CPU usage, etc. Those details allow us to peek into the intrinsic features of those DL libs and therefore provide more insightful implications to developers and contribute to a more useful, and less fragmented Web of Things.

Based on our benchmark, we perform extensive experiments to demystify the performance of DL libs on various models (15 in total) and hardware (10 smartphones that are equipped with CPU/GPU/DSP). Through the experiments, we make the following interesting and useful observations.

(1) The performance of the 6 DL libs benchmarked is severely fragmented across different models and hardware (§3.1). There is no **One-Size-Fits-All** DL lib that performs best on all scenarios (model⊗device), yet each DL lib has at least one best-performing scenario. Even for the same model, there are different DL libs that perform the best on different devices. The performance gap between those DL libs is huge, i.e., about 7.4×/1.9× among the best and the worst/2nd-best ones on average. On mobile GPU, such fragmentation is further exaggerated by the multiple choices of software driver backends (Vulkan/OpenGL/OpenCL).

(2) The impacts of DL software may overwhelm the algorithm designs and hardware capacity (§3.2, §3.3). Designing a more lightweight model structure, model quantization (FP32 to INT8), and using mobile GPUs/DSPs with high parallelism are common ways to speed up on-device inference. However, due to the defects from DL libs, those methods cannot always bring expected benefits or even slow down the inference. For instance, INT8-based quantization only brings 0.8×–3.0× inference speedup (≤1 means slowdown), which is much less than the theoretical expectation, i.e., 4× due to the NEON support in Android [36].

(3) There is a noteworthy potential to further enhance the DL lib performance by integrating the optimal implementation of different DL libs at operator level (§3.4). Motivated by the severe fragmentation of DL libs, we perform an emulation test assuming that we can combine the best of each DL libs at operator level. Through such combinations, we can potentially reduce the inference time by up to 29.9% compared to the best-performing lib.

(4) Cold-start inference of DL libs is significantly slower than warm inference (§3.5). On average, the first inference for each session is about 10.8× and 25.7× slower than the following ones on CPU and GPU, respectively. Diving deeper, we find that the memory preparation stage contributes to the most of the overhead, which includes expanding the loaded weights to proper memory locations and reserving memory for intermediate feature maps.

(5) During the evolution of DL libs, performance bugs are introduced for many times (§3.6). By benchmarking the weekly version of `TFLite` and `ncnn` since early 2018, we find that the overall performance of DL libs is improving yet becomes relatively stable since 2020. Surprisingly, we observe that some commits incur significant performance degradation on certain scenarios, which can take 1–16 weeks to be fixed. For example, some commits provide new forms of operator implementation, aimed to improve the inference performance. Those changes, however, result in performance degradation in certain models or devices.

Atop the above observations, we summarize the key implications to different roles in the mobile DL ecosystem.

For DL app/model developers: (i) It is extremely challenging in selecting a proper DL lib due to the severe fragmentation. To pursue the optimal performance under each scenario, they have to embed different DL libs into the apps and load one dynamically based on the model and hardware settings. (ii) A more lightweight model (fewer FLOPs) does not always run fast. The impact from software at deployment needs to be considered during model design.

For DL lib engineers and researchers: (i) It is time to review the pros and cons of different DL libs and work out a solution that can integrate their wisdom in a unified manner. Otherwise, the fragmentation may continuously exist for a long term as fixing it can take huge amount of engineering efforts. (ii) The cold-start inference time is a rarely touched topic, but can be important in apps that only need to execute models for one time per session. Potential optimizations include using multi-thread to speed up memory preparation and operator-level pipeline of different initialization stages. (iii) Performance bugs bring negative impacts to the open-source ecosystem of DL libs, but are difficult to be fully eliminated due to the aforementioned fragmentation. Tools that can automatically identify such bugs timely, either through dynamic or static analysis, are urgently needed.

Our main contributions are as follows.

- We design and implement `MDLBench`, a fully automatic, comprehensive benchmark for DL libs. The full benchmark suite is open-sourced[1].
- We perform extensive experiments with `MDLBench` on diverse hardware devices and models, and the results reveal a complete landscape of the current DL lib ecosystem.
- We summarize the insightful observations and practical implications based on the experiments that can benefit different roles in the DL lib ecosystem.

2 Benchmark and Methodology

`MDLBench` is a benchmark aimed to understand the impacts of DL libs on the on-device DL performance. It has the following advantages over existing AI benchmarks.

• **Rich support** Table 2 summarizes the DL libs (6 in total), models (15 in total), and hardware processor (CPU/GPU/DSP) `MDLBench` currently supports. Being able to test many DL libs under various contexts is critical to obtain a complete landscape of the DL lib ecosystem, because the performance optimization is quite ad-hoc to models and hardware. Among the large amount of DL libs available for developers, we select 6 most representative ones from a "market" perspective. We follow the prior works [72] to detect the DL libs used in 16,000 Android apps we crawled in Mar. 2021 from Google Play. Among the 676 DL apps identified, we find the most popular DL libs are TFLite (70.5%), TensorFlow (7.8%), ncnn (7.2%), caffe (4.4%), MNN (4.1%), PyTorchMobile (3.8%), Mace (1.2%). We filter TensorFlow and caffe, as their support for smartphones are deprecated a few years ago and has been merged into the corresponding lightweight implementation, i.e., TFLite and PyTorchMobile. We further include SNPE into `MDLBench`, as it's a vendor-specific (Qualcomm) DL lib while all above are not. The models we use come from two sources. One is the model zoo of TensorFlow and PyTorch [9, 11]. The other is by using the built-in converters of each

[1] https://github.com/UbiquitousLearning/MobileDLFrameworksBenchmark

Models	Tasks	TFLite	ncnn	mnn	MACE	PyTorchMobile	SNPE
mobilenetV1[29]	image classification	$C_{32,8}$-$G_{32,8}$-D_8	$C_{32,8}$-$G_{32,8}$	$C_{32,8}$-$G_{32,8}$	$C_{32,8}$-G_{32}	$C_{32,8}$	$C_{32,8}$-$G_{32,8}$-D_8
mobilenetV2[55]	image classification	$C_{32,8}$-$G_{32,8}$-D_8	$C_{32,8}$-$G_{32,8}$	$C_{32,8}$-$G_{32,8}$	$C_{32,8}$-G_{32}	$C_{32,8}$	$C_{32,8}$-$G_{32,8}$-D_8
inceptionV3 [60]	image classification	$C_{32,8}$-$G_{32,8}$-D_8	$C_{32,8}$-$G_{32,8}$	$C_{32,8}$-$G_{32,8}$	C_{32}-G_{32}	$C_{32,8}$	$C_{32,8}$-$G_{32,8}$-D_8
inceptionV4 [59]	image classification	$C_{32,8}$-$G_{32,8}$-D_8	$C_{32,8}$-$G_{32,8}$	C_{32}-G_{32}	C_{32}-G_{32}	$C_{32,8}$	$C_{32,8}$-$G_{32,8}$-D_8
vgg16 [57]	image classification	$C_{32,8}$-$G_{32,8}$-D_8	$C_{32,8}$-$G_{32,8}$	$C_{32,8}$-$G_{32,8}$	C_{32}-G_{32}	$C_{32,8}$	$C_{32,8}$-$G_{32,8}$-D_8
squeezenet [32]	image classification	$C_{32,8}$-$G_{32,8}$-D_8	$C_{32,8}$-$G_{32,8}$	C_{32}-G_{32}	C_{32}-G_{32}	$C_{32,8}$	$C_{32,8}$-$G_{32,8}$-D_8
nasnet_mobile [84]	image classification	C_{32}-G_{32}	-	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}	-
densenet [31]	image classification	C_{32}-G_{32}	-	C_{32}-G_{32}	-	C_{32}	C_{32}-G_{32}
mnasnet [61]	image classification	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}	C_{32}-G_{32}
resnetv2_50 [58]	image classification	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}	C_{32}-G_{32}
deeplabv3 [81]	semantic segmentation	C_{32}-G_{32}	-	C_{32}-G_{32}	C_{32}-G_{32}	-	-
ssd_mobilenetV1 [46]	object detection	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}	-
yolo-fastest [80]	object detection	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	-	-	-
yolo3 [53]	object detection	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	-	-	-
albert_tiny [70]	text classification	C_{32}-G_{32}	-	C_{32}-G_{32}	-	-	-

Table 2: The supported DL libraries and models of MDLBench. "C/G/D": mobile CPU/GPU/DSP. The subscripted 32 and 8 represent different model precision, i.e., float32 and int8, respectively.

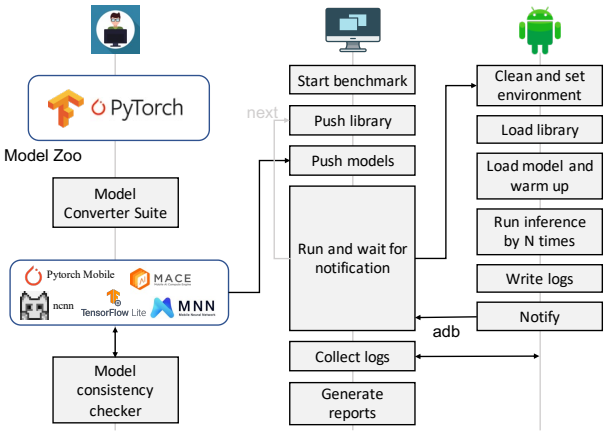


Figure 1: Workflow of MDLBench.

DL lib to convert models to different formats [2–5]. MDLBench also incorporates a module to automatically check the equivalence of the same model generated for different DL libs.

• **Detailed metrics** MDLBench profiles the inference time and operator-level information, e.g., per-operator latency, duration, input/output dimension, etc. Such functionality is originally supported for some DL libs (TFLite, SNPE) yet for others (MNN) we need instrumenting the source code. MDLBench then adds another layer of traces processing to unify and visualize the output from different DL libs.

Workflow Figure 1 shows the overall workflow of MDLBench. For each testing, the desktop-side benchmark iterates over each DL lib. It first pushes the lib and corresponding models generated as aforementioned to the devices through adb [38]. Next, the device cleans the system environment by killing other apps in background and sets the system configurations (CPU frequency, thread number, etc). Following prior work [23, 66], we always use 4 big cores to run the DL libs as it's often the best-performing setting. The device then loads the library and model into memory to warm up, and executes the inference by N times (50 by default). The testing results will be written to device storage and retrieved to desktop.

Devices Table 3 shows the devices used in our measurement. It includes 10 different device models with various SoCs (Snapdragon series, Kirin, Helio) and GPUs (Adreno series and Mali series), where the currently selected smartphones are still representative to reflect the hardware heterogeneity.

Devices	abbr.	SoC	GPU	RAM
Google Pixel5	GP5	Snapdragon 765G	Adreno 620	8GB
Huawei Enjoy 8	HE8	Snapdragon 430	Adreno 505	4GB
MeiZu 16T	MZ16	Snapdragon 855	Adreno 640	6GB
HuaWei Mate30	HM	Kirin 990	Mali-G76 MP16	8GB
XiaoMi11 Pro	MI11	Snapdragon 888	Adreno 660	8GB
XiaoMi9	MI9	Snapdragon 855	Adreno 640	6GB
MeiZu 16T	MZ16	Snapdragon 855	Adreno 640	6GB
OnePlus 9R	OP9	Snapdragon 870	Adreno 650	8GB
RedMi9	R9	Helio G80	Mali-G52 MC2	4GB
Redmi Note9 Pro	RN9	Snapdragon 720G	Adreno 618	6GB
Samsung S21	S21	Snapdragon 888	Adreno 660	8GB

Table 3: The tested devices and their specifications.

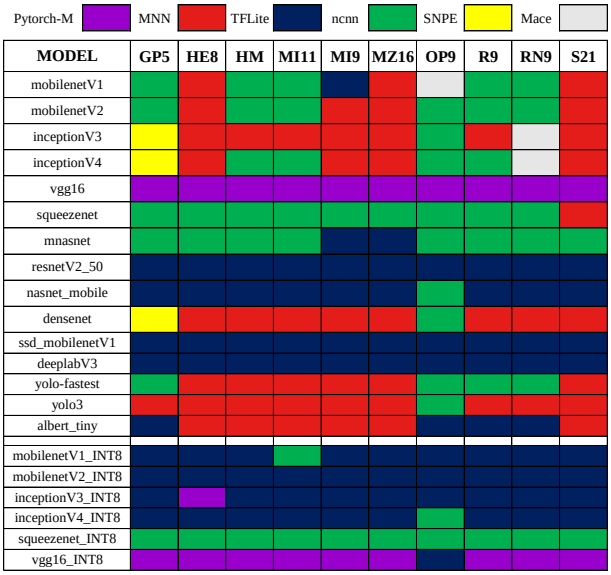
3 Performance Analysis

Based on MDLBench and the diverse mobile devices, we perform extensive experiments and analyze the results. The theme of this measurement is to quantitatively understand the performance discrepancy of different DL libs, and how the inter-play with the impacts from algorithm and hardware. Besides, we also explore two rarely touched topics in mobile community: what is the performance of the first inference (cold start) of different DL models, and how does the performance of DL libs evolve across time.

3.1 Performance Fragmentation

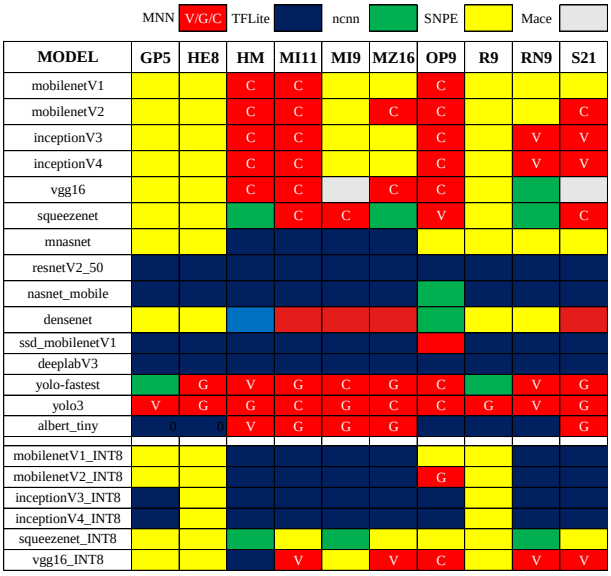
Figure 2 summarizes the best-performing DL lib (by color), i.e., the DL lib with the smallest inference time when running different models on heterogeneous devices. We observe that the performance of DL libs across models and hardware devices is severely fragmented.

(1) **There is no one-size-fit-all DL lib for optimal performance across models and hardware.** On either CPU and GPU, a silver-bullet best-performing DL lib does not exist. Different DL libs have different adept scenarios (model × device), Table 4 showcases the number of best performing scenarios of different DL libs, as summarized from Figure 2. Each DL lib has at least one best-performing



MODEL	GP5	HE8	HM	MI11	MI9	MZ16	OP9	R9	RN9	S21
mobilenetV1										
mobilenetV2										
inceptionV3										
inceptionV4										
vgg16										
squeezenet										
mnasnet										
resnetV2_50										
nasnet_mobile										
densenet										
ssd_mobilenetV1										
deeplabV3										
yolo-fastest										
yolo3										
albert_tiny										
mobilenetV1_INT8										
mobilenetV2_INT8										
inceptionV3_INT8										
inceptionV4_INT8										
squeezenet_INT8										
vgg16_INT8										

(a) CPU



MODEL	GP5	HE8	HM	MI11	MI9	MZ16	OP9	R9	RN9	S21
mobilenetV1			C	C			C			
mobilenetV2			C	C		C	C			C
inceptionV3			C	C			C		V	V
inceptionV4			C	C			C		V	V
vgg16			C	C		C	C			
squeezenet				C	C		V			C
mnasnet										
resnetV2_50										
nasnet_mobile										
densenet										
ssd_mobilenetV1										
deeplabV3										
yolo-fastest		G	V	G	C	G	C		V	G
yolo3	V	G	G	C	G	C	C	G	V	G
albert_tiny	0	0	V	G	G	G				G
mobilenetV1_INT8										
mobilenetV2_INT8							G			
inceptionV3_INT8										
inceptionV4_INT8										
squeezenet_INT8										
vgg16_INT8				V		V	C		V	V

(b) GPU. The characters in MNN indicate different GPU backends: V/G/C indicate Vulkan/OpenGL/OpenCL.

Figure 2: The best-performing DL lib (smallest inference time) with different models and devices.

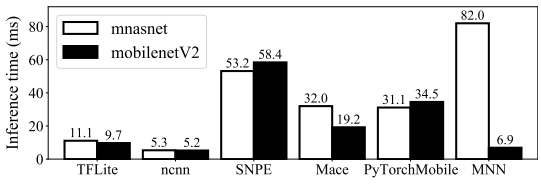


Figure 3: The average inference time of squeezenet and mnasnet with different libs on MI11.

scenario, except that PyTorchMobile does not support GPU acceleration. Even for the same model, its corresponding best-performing

	TFLite	PyTorchMobile	ncnn	MNN	Mace	SNPE
CPU	84	20	52	48	3	3
GPU	80	/	12	49	5	64

Table 4: The number of best-performing scenario of each DL lib, summarized from Figure 2. A scenario is defined by a pair of DL model and a tested device.

Models	Best vs. Worst		Best vs. 2nd Best	
	CPU (×)	GPU (×)	CPU (×)	GPU (×)
mobilenetV1	4.0~15.4 (8.7)	1.7~14.1 (5.6)	1.1~1.9 (1.5)	1.0~4.0 (1.9)
mobilenetV2	5.6~18.8 (11.2)	2.9~15.9 (6.2)	1.1~2.0 (1.5)	1.0~2.9 (1.6)
inceptionV3	2.6~5.6 (3.8)	3.0~13.4 (7.1)	1.1~2.4 (1.7)	1.0~4.0 (2.1)
inceptionV4	2.0~5.4 (3.2)	2.4~11.0 (5.8)	1.1~2.0 (1.5)	1.0~3.6 (2.0)
vgg16	7.1~54.3 (16.2)	4.4~7.0 (5.5)	1.3~4.2 (2.4)	1.1~2.2 (1.5)
squeezenet	4.6~19.9 (9.1)	1.9~12.6 (5.9)	1.0~5.9 (2.5)	1.1~2.5 (1.6)
average	8.7	6.0	1.9	1.8

Table 5: The performance gaps of different DL libs.

DL lib may change across different hardware. For instance, the best-performing DL libs of inceptionV3 are SNPE, ncnn, and Mace on GP5, OP9, and RN9, respectively.

Such high performance fragmentation mainly attributes to two facts. First, mobile hardware ecosystem is highly fragmented in consideration of their Big.Little Core architecture, cache size, GPU capacity, etc. Second, the model structure is also heterogeneous. Implementing depth-wise convolution operator [27] is totally different from traditional convolution operators as they have different cache access patterns. Overall, the fragmentation of models and hardware forces the software-level DL inference optimization especially model- and hardware-specific. To obtain the optimal performance, DL lib developers need to hand-craft each operator at very low-level programming interface, heavily relying on assembly language and NEON instructions. While being able to fully exploit the capacity of specific hardware, such implementation cannot be generalized well to different hardware platforms. For example, ncnn has 44 different types of implementation for convolution operation, each fitting to different execution contexts like kernel size, hardware architecture, etc. Due to the high manual programming efforts, there is no oracle DL lib optimized for each scenario.

(2) **The performance gap of DL libs can be large.** To show the absolute numerical gap between DL libs performance, Table 5 summarizes the performance gap of 6 models between the best-performing DL libs and the worst/2nd-best. The "gap" is defined as the ratio of inference time of two DL libs (the longer one divided by the shorter one). The numbers in parentheses are average values. Surprisingly, though those DL libs are all specifically designed and optimized for mobile devices, the performance gap can be quite severe. For instance, for the same model vgg16, the gap between different libs and smartphones is as high as 54.3×, and even the smallest gap between the best and the second best is 1.5×. On average, the gap between the best-performing to the worst one is 7.4×, and to the 2nd-best one is 1.9×.

(3) **GPU backend choices further exaggerate the fragmentation.** Even on the same GPU, there are different backend choices implemented by DL libs. For example, MNN implements three backends: Vulkan, OpenGL and OpenCL [36, 49, 56]. Interestingly, as

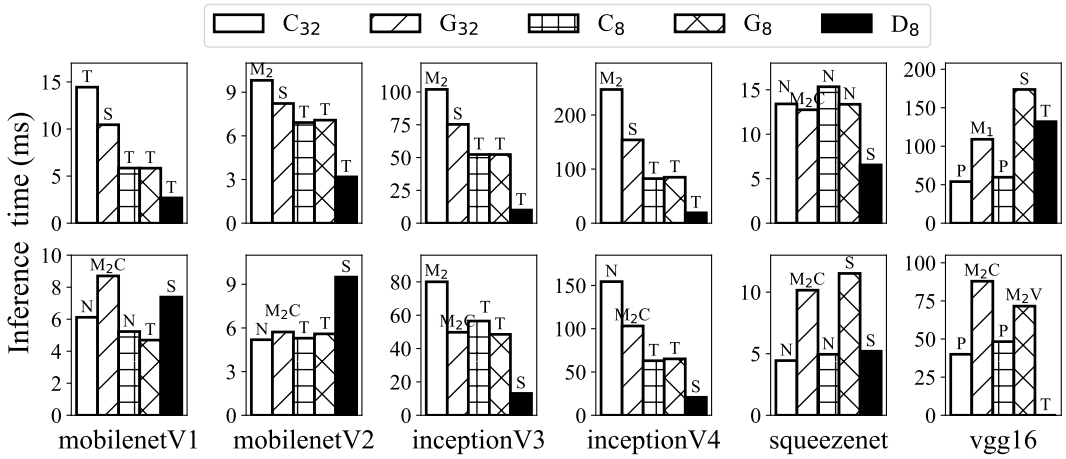


Figure 4: The best inference speed across all DL libs. "T", "N", "S", "P", "M_1", and "M_2" are short for `TFLite`, `ncnn`, `SNPE`, `PyTorchMobile`, `Mace`, and `MNN` respectively as the best-performing DL libs. "V", "C", "G" indicate different GPU backends. We leave out vgg16 with SNPE since the model does not work on MI11.

shown in Figure 2(b), different GPU backend choices also fit different models and devices. This is somehow surprising because Vulkan in `MNN` is mainly used for cross-platform compatibility (e.g., desktop), while OpenGL/OpenCL are mobile-specific programming interfaces highly optimized for mobile devices [49]. Such phenomenon attributes to both the underlying design of backends and how DL developers implement the DL operators atop the backends.

(4) **With software heterogeneity, the model structure is not the sole factor that determines their relative performance.** We deem that model complexity does affect the inference time, e.g., the computation complexity represented by floating-point operations (FLOPs) and the number of models parameters. In fact, the complexity can also be affected by the structural heterogeneity, since heterogeneity makes on-device optimization more difficult. For example, although mobilenetV2 and mnasnet have similar FLOPs (300 million vs. 315 million) and parameters (3.4 million vs. 3.9 million), their performances vary a lot across DL libs. As shown in Figure 3, squeezenet runs faster than mobilenetV2 with `SNPE`, `PyTorchMobile`, while mobilenetV2 runs faster with other DL libs.

†**Implication** *The best-performing DL lib is highly fragmented across models and hardware. Such fragmentation may even overwhelm the model designs and hardware capacity improvement. To pursue the optimal performance in a mobile DL app, the developers need to incorporate different DL libs and dynamically load one based on the current model and hardware platform. Such a methodology is rarely seen in practice as it incurs significant overhead to both software complexity and developing efforts. A more lightweight system is desired to bring together the best performance of different DL libs.*

3.2 Impacts of Quantization

Quantization has become a common practice to deploy DL models on mobile devices. There are different levels of quantization, e.g., FP16, INT16, INT8, etc [28, 64]. Among them, INT8-based quantization is known to achieve the best trade off among model accuracy and on-device speedup. Therefore, we mainly study INT8-based performance on CPU/GPU/DSP.

Benefit brought by INT8 quantization is under expectation. Figure 4 summarizes the best inference performance across DL libs on different model representations and hardware. It shows that quantization indeed brings inference speedup in most scenarios. However, the speedup (0.8×–3.0×) is much less than the theoretical expectation (4× due to the NEON support in Android [36]). In certain cases, the INT8-based inference is even slower than FP32, e.g., with squeezenet and vgg16 on M11 CPU. Furthermore, whether quantization can accelerate model inference also relies on the underlying hardware, i.e., the SoCs and the processor.

We dive into the source code of those DL libs and identify the following reasons. (1) Modern mobile SoCs also have good support for FP processing. (2) FP32-based tensor operations are better tunned than INT8, according to our observations to the commit history of those DL libs. (3) Overhead of converting between INT8 and FP32 can incur nontrivial overhead. For example, re-quantization is essential in the final softmax layer of most classification models.

†**Implication** *Not every model can be accelerated through INT8 quantization, and the situation may vary across different hardware devices and processors. There exists great potential at software level to accelerate the inference of quantized models.*

3.3 Impacts of Hardware

We then investigate whether and to what extent can more powerful CPUs or heterogeneous processors (GPU/DSP) on smartphones can accelerate DL inference. The results are shown in Figure 4.

Newer generations of mobile SoCs can mostly accelerate the inference, yet not in every case. As the most representative SoC series of mobile devices, new generation of Qualcomm Snapdragon comes out every one or two years. As shown in Figure 5, from the Snapdragon 430 to 888, the overall performance of the three libraries (`TFLite`, `MNN`, `SNPE`) shows a similar trend of improving. However, there are cases when newer SoC runs slower than the

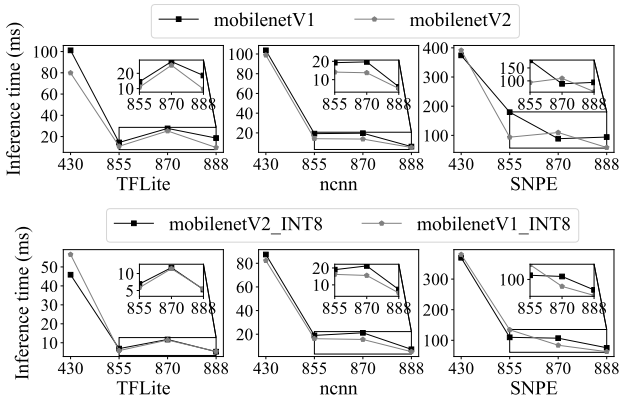


Figure 5: The performance across different SoCs.

old ones, e.g., Snapdragon 870 vs. 855 on `TFLite`, even though 870 is equipped with stronger CPU and faster memory access speed [10]. This is mainly because Snapdragon 855 is a more popular SoC for which the DL libs are highly optimized.

GPUs can not always accelerate DL inference. For most cases of FP32-based models, GPU can indeed bring inference speedup by 1.4×–1.9× compared to CPU. However, in certain cases like mobilenetV1 and vgg16 on MI11, GPU even runs slower than CPU (up to 2.3×). On INT8-based models, GPU can hardly bring any benefit.

There are following primary reasons. Firstly, mobile GPUs are mainly designed for rendering instead of general-purpose computing. Their computing power is highly constrained due to the battery life consideration [25]. Secondly, the DL libs are not as well optimized for GPUs as CPUs. During experiments, we observe that the arithmetic processing units inside GPU cores are often underutilized. Thirdly, mobile GPUs often do not have native support for INT8 data format, therefore the actual inference falls back to FP32. Fourth, there lack GPU support for some operators (e.g., SQUEEZE on TFLite), and those operators will fall back to run on CPUs, which incurs nontrivial overhead for data copy among CPU and GPU[2].

†**Implication** *Our findings motivate DL lib developers to focus on GPU-side optimization [66], including supporting more types of operators and single-op performance. It also motivates DL researchers to design the models suitable for GPU computing, that is, the operators in the models with a large number of parallel features as much as possible, and reduce high memory access operators that are not good for parallel operations.*

DSP can significantly accelerate INT8 model in most cases. Figure 4 also shows that running on mobile DSP can reduce the inference time of INT8 model by 2.0×–12.9×. This is mainly because Qualcomm DSP has been equipped with AI capabilities such as HTA and HTP [17], which are integrated with Hexagon vector extension (HVX) acceleration. Meanwhile, the Winograd algorithm is used to accelerate the convolution calculation on DSP. In fact, the energy saving of DSP is even more significant than inference speed (not shown in the Figure) according to our measurements.

However, there are a few cases that DSP performs worse than CPU (mobilenetV1/V2 on MI11). This is mainly because MI11 uses Snapdragon 888 SoC, which is a relatively new chip that the DL libs are not currently well tuned for.

[2]Though mobile CPU and GPU share the same memory unit, their memory spaces are separated by OS and cannot be accessed mutually.

Models	Mace	tflite	SNPE	ncnn	Oracle time
mobilenetV1	19	18.3	50.3	14.4	13.5 (↓6.1%)
mobilenetV2	37.9	31.5	113.4	14.4	10.6 (↓26.3%)
inceptionV3	230	154.9	176.7	123	86.3 (↓29.9%)
inceptionV4	293	195.1	312.6	374.9	180.3 (↓7.6%)
vgg16	180.3	73.1	341.7	409.0	73.1 (↓0%)

Table 6: The benefits of an "oracle DL lib" that integrates the best-performing operator of all tested DL libs. The numbers in parenthsis indicates the potential improvement over the best-performing DL lib (different for each model).

†**Implication** *In most cases, more powerful CPUs and accelerators (GPU and especially DSP) can speed up the model inference. However, there are cases that DL libs perform even worse on those hardware. In other words, the current DL libs can not fully exploit the capacity of each hardware. Our findings motivate DL lib developers to focus on optimization on heterogeneous processors [66], including supporting more types of operators and single-op performance. It also motivates DL researchers to design the models suitable for GPU computing and reduce high memory access operators that are not good for parallel operations.*

3.4 Operator-level Integration of DL Libs

Motivated by the severe fragmentation of DL libs on diverse models and hardware, we then explore the idea of "how much benefits can be brought if we can integrate operator-level wisdom from different DL libs". More specifically, we collect the per-operator inference time for each DL lib, and combine the best-performing DL lib for each operator. Therefore, we obtain an "oracle lib" that combines the fastest operator from those DL libs.

Table 6 summarizes the performance that can be achieved by such oracle lib. In summary, by integrating the operator implementation of different DL libs, we can achieve 0%–29.9% inference time reduction compared to the best-performing DL lib across different models. Such improvement is nontrivial as the best-performing DL lib is already highly optimized for specific models and hardware, and their performance has reached a stable point in recent two years as shown in §3.6.

We use inceptionV4 as an example to show how different DL libs perform at operator level. We find `Mace` has best support for convolution with 3x3 kernels, while `ncnn` performs best for convolution with 1x7 and 1x3 kernels. For some activation and software layers, `TFLite` performs best.

†**Implication** *Our findings further highlight the fragmentation of DL libs at operator level, and motivate that by combining the wisdom from different libs, the DL inference performance can be further upgraded. However, achieving such potential is not easy, as differnt DL libs have different ways to implement operators, e.g., the tensor alignment and memory pool. Those diversities need to be unified before the operator implementation can be combined.*

3.5 Cold-start Inference

The above results are all based on "warm" execution, i.e., the continuous inference after the first 5 rounds of inference. However,

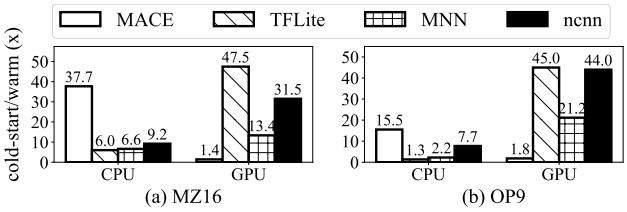


Figure 6: The ratio of cold-start inference to warm inference. Numbers are averaged across all DL models.

"cold-start" inference, i.e., the first inference beginning from model loading, is also important because for many apps the inference only happens once. In addition, cold-start inference is also important when apps expectedly crash and need to recover the DL functionality as fast as possible.

Cold-start inference is significantly slower than warm inference. Figure 6 shows how much times (×) slower cold-start inference is on CPU and GPU averaged across all models on two mobile devices. Overall, cold-start inference is much slower than warm inference, i.e., 1.3×–37.7× on CPU and 1.4×–45.0× on GPU.

Memory preparation contributes to the largest overhead in cold-start inference. To investigate the reasons of slow cold start, we dive into the source code of ncnn and identify the workflow of the cold-start inference. It consists of three major steps: loading model from disk, memory preparation, and running inference. The memory preparation main refers to expanding the loaded weights to proper memory locations and reserving memory for intermediate feature maps to speed up the later inference. For example, both img2col [44] and Winograd [69] implementation of convolution operation require to transform the original convolution kernel matrix to a different shape.

Figure 7 quantitatively shows the breakdown of cold-start inference of ncnn on 5 models and 2 devices. As observed, memory preparation is the one that accounts for the largest proportion of cold-start inference of all models, i.e., 67% on CPU and 91% on GPU on average. In fact, we observe that memory preparation is implemented in a single thread in ncnn and other DL libs, therefore cannot benefit from the multi-core system of mobile SoCs. Additionally, memory preparation for GPU inference even takes more time than on CPU because of the complicated model, i.e., the code needs to be compiled to shader before executing on GPU [63].

†**Implication** *Optimization of cold-start inference is a rarely explored topic, but can be important in many apps that only need to execute model once each time. Potential solutions include speeding up memory preparation using multiple threads and generating pipeline to run model loading (I/O-intensive), memory preparation (memory-intensive), and inference (compute-intensive) simultaneously.*

3.6 Longitudinal Analysis

We then longitudinally analyze how the performance of DL libs evolves across time. We select 2 DL libs that have the longest open source history and test their performance on the commits at the beginning of every week from Mar./Jul. 2018 to Jul. 2021 (80,637 commits in total) respectively. For simplicity, we only show test models (mobilenetV1/V2 and squeezenet) on CPU and GPU.

Overall, the performance of DL libs are continuously improving in early years, but becomes relatively stable since

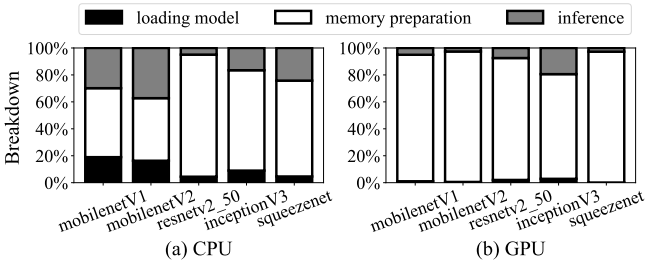


Figure 7: The breakdown of cold-start inference time.

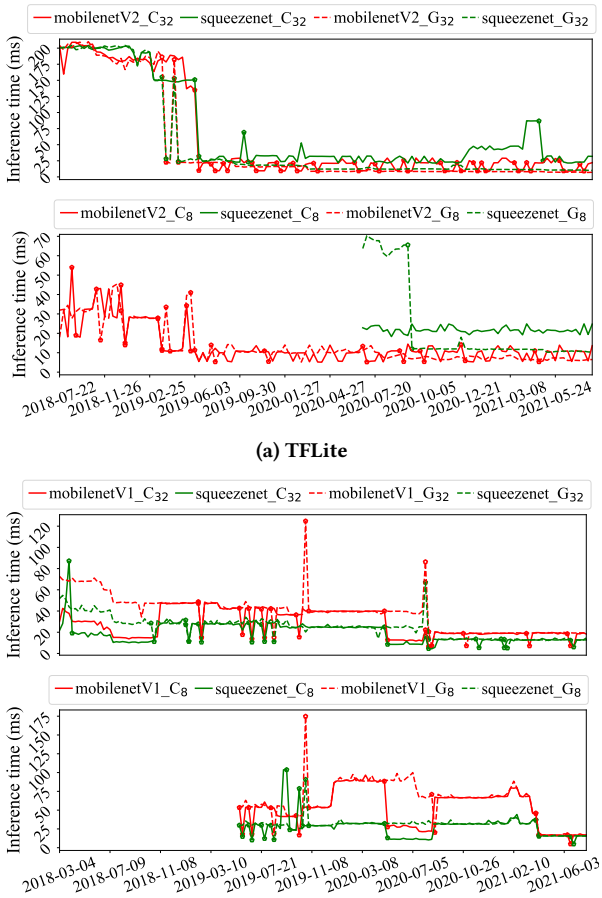


Figure 8: The inference performance evolvement across time of TFLite & ncnn tested on HM device.

2020. As shown in Figure 8, the performance of TFLite and ncnn are improving: taking mobilenetV2 (FP32 format) as an example, its inference time on CPU/GPU has reduced from 203.6ms/203.8ms to 21.9ms/7.2ms with TFLite, and 30.3ms/72.7ms to 19.5ms/19.7ms with ncnn, respectively. Similar observation is also made on squeezenet and the INT8 models. The performance improvement is mostly a cliff-like change in a few commits, rather than a regular and slow change. However, since 2020, the performance of DL libs is relatively stable and there are very few nontrivial improvements. It indicates that the DL lib community is shifting their focus from

performance optimization to other aspects, e.g., supporting more types of operators.

We also observe that a commit may only improve the performance of certain models. For example, the *20275fe* commit on TFLite in Jun. 6, 2019 reduces the inference time of mobilenetV2 by 13.6×, but hardly affects the inference time of mobilenetV2. The reason of such "partial improvement" is the same as the fragmentation of DL libs as mentioned in §3.1.

In certain commits, we observe significant performance degradation. For instance, the *20cd7182* commit on TFLite in Jul. 8, 2020 increases the inference time of from 9.3ms to 21.9ms. Similar phenomenon also exists in ncnn: the *971fe2f* commit on ncnn in Nov. 3, 2018 increases the inference time of squeezenet from 11.6ms to 24.8ms. We regard such commits that cause significant performance degradation on certain models and devices as *performance bug*. We dig into those commits' contents, and find that except "real bugs", a common reason is that a new operator implementation is pushed to improve the performance of DL models, but do not work well under certain settings, which is unexpected to the commit submitter. For instance, a convolution kernel may perform well with 3x3 kernel size and stride size 2, but not with 3x3 kernel size and stride size 1. Among the 34 detected performance bugs, it takes around 1–16 weeks to fix so that the performance can be recovered.

†**Implication** *The current open-source ecosystems of DL libs sometimes introduce performance bugs, possibly due to a comprehensive benchmarking tool available for developers to test their commits. Indeed, due to the performance heterogeneity of DL libs on different models and hardware, it is almost impossible to fully eliminate performance bugs. We propose two possible solutions. One is to set up an environment with diverse device models periodically (e.g., per day) running a comprehensive benchmark like* MDLBench *to timely detect performance bugs. Another one is to build a static analysis tool that can identify commits with potential bugs based on history.*

4 Related Work

Mobile DL In recent years, there is a notable trend to move DL inference into local devices instead of offloading to remote servers [41, 43, 73–77]. A fundamental challenge of this trend is the constrained resources of smartphones. Therefore, performance optimization has been a primary research direction for both academia and industry [18, 39, 40, 50, 71, 79, 82]. There have been some optimization research efforts addressed to recude the overhead of DL on smartphones, e.g., offloading, model quantization and sparsity [35, 37, 45, 68]. These solutions usually either count on preprocessing or perform under lab simulations on the data collected preciously from smartphones. Thus, our work brings DL to smartphones in the real world and provides a unified approach to easily compare performance among different libs. This work is motivated by many years of efforts at this lane.

AI benchmarks As summarized in Table 1, there exist a few AI benchmarks for diversified scenarios, e.g., datacenter servers or edges, inference or training, etc [13, 15, 16, 19, 22, 34, 48]. This work explicitly targets at inference on mobile devices. Besides, the ecosystem of on-device DL libs is more fragmented than servers due to the high fragmentation of mobile hardware. Furthermore, a number of studies focus on DL libs analysis. Consequently, the results are limited in small-scale project from the specific perspective. Luo et al. [47] proposed the benchmark suite for evaluating the abilities of mobile devices across different libs. MLPerf [48] proposed high-level rules for more flexible benchmark of the libs. Tang et al. [62] studied the behavior characteristics of neural networks to bridge networks design and real-world performance. There is still limited understanding about the performance of DL libs across heterogeneous smartphones. Compared to similar benchmarks focusing on DL libs, MDLBench has richer support for various DL libs and models.

Empirical study of mobile DL One line of studies mainly focus on DL apps/systems/models. Xu et al. [72] demystified how smartphone apps exploit DL models by deeply analyzing Android apps. Wang et al. [65] made efforts towards the evolution of mobile app ecosystem. Andrey et al. [33] targeted at devices and focused on running models with hardware acceleration of smartphones. Although the studies have analyzed on device DL, they lack a comprehensive understanding and benchmarking on diverse libs.

Deep learning compilers [20, 26, 83] represent a different way to deploy DL models compared to static libs. The key idea of DL compilers is to pre-define primitives of operators and rules to find an optimal implementation. Our benchmark does not include such compilers because of following reasons. First, the state-of-the-art DL compilers like TVM [24] are mainly designed for servers instead of mobile devices. According to our measurements, on most of models, the resultant performance is not even close to our tested DL libs after many hours of search. Second, because of the high fragmentation of mobile devices, generating an execution plan for each device is impractical. In fact, some DL libs already search for an optimal working group size in their GPU implementation.

5 Conclusions

In this work, we built the first comprehensive benchmark for DL libs and conducted extensive measurements to quantitatively understand their performance. The results help reveal a complete landscape of the DL libs ecosystem. Atop the observations, we summarize strong implications that can be useful to developers and researchers.

Acknowledgments

This work was partly supported by National Key R&D Program of China under grant number 2020YFB1805500, and National Natural Science Foundation of China under grant number 61922017 and 62102009. Mengwei Xu was sponsored by CCF- Baidu Open Fund. Xuanzhe Liu was supported by PKU-Baidu Fund Project under the grant number 2020BD007 and Alibaba Group through Alibaba Innovative Research (AIR) Program.

References

[1] Performance measurement | TensorFlow Lite. https://www.tensorflow.org/lite/performance/measurement.
[2] Snapdragon snpe deep learning framework. https://developer.qualcomm.com/sites/default/files/docs/snpe/overview.html, 2017.
[3] Xiaomi mace deep learning framework. https://github.com/XiaoMi/mace, 2017.
[4] Tencent ncnn deep learning framework. https://github.com/Tencent/ncnn, 2018.
[5] Alibaba mnn deep learning framework. https://github.com/alibaba/MNN, 2019.
[6] Pytorch mobile. https://pytorch.org/mobile/home/, 2019.
[7] Ai matrix: To make it easy to benchmark ai accelerators. https://github.com/alibaba/ai-matrix, 2020.

[8] Deep learning market - growth, trends, forecasts (2020-2025). https://www.mordorintelligence.com/industry-reports/deep-learning, 2020.

[9] Pytorch model zoo. https://pytorch.org/serve/model_zoo.html, 2020.

[10] Qualcomm snapdragon 855 vs qualcomm snapdragon 870. https://versus.com/en/qualcomm-snapdragon-855-vs-qualcomm-snapdragon-870, 2020.

[11] Tensorflow model zoo. https://github.com/tensorflow/models, 2020.

[12] White paper : On artificial intelligence - a european approach to excellence and trust. https://ec.europa.eu/info/sites/default/files/commission-white-paper-artificial-intelligence-feb2020_en.pdf, 2020.

[13] Aiia dnn benchmark. https://github.com/AIIABenchmark/AIIA-DNN-benchmark, 2021.

[14] Artificial intelligence market analysis report. https://www.grandviewresearch.com/industry-analysis/artificial-intelligence-ai-market, 2021.

[15] Dawnbench: An end-to-end deep learning benchmark and competition. https://dawn.cs.stanford.edu/benchmark/CIFAR10/inference.html, 2021.

[16] Deepbench: Benchmarking deep learning operations on different hardware. https://github.com/baidu-research/DeepBench, 2021.

[17] Qualcomm hexagon. https://en.wikipedia.org/wiki/Qualcomm_Hexagon, 2021.

[18] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI)*, pages 265–283, 2016.

[19] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Fathom: Reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[20] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. *arXiv preprint arXiv:2001.08743*, 2020.

[21] Mario Almeida, Stefanos Laskaridis, Ilias Leontiadis, Stylianos I Venieris, and Nicholas D Lane. Embench: Quantifying performance variations of deep neural networks across modern commodity devices. In *The 3rd international workshop on deep learning for mobile systems and applications*, pages 1–6, 2019.

[22] Mario Almeida, Stefanos Laskaridis, Abhinav Mehrotra, Lukasz Dudziak, Ilias Leontiadis, and Nicholas D Lane. Smart at what cost? characterising mobile deep neural networks in the wild. *arXiv preprint arXiv:2109.13963*, 2021.

[23] Dongqi Cai, Qipeng Wang, Yuanqiang Liu, Yunxin Liu, Shangguang Wang, and Mengwei Xu. Towards ubiquitous learning: A first measurement of on-device training performance. In *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, pages 31–36, 2021.

[24] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 11:20, 2018.

[25] Sofiane Chetoui and Sherief Reda. Workload-and user-aware battery lifetime management for mobile socs. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1679–1684. IEEE, 2021.

[26] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. Boat: Building auto-tuners with structured bayesian optimization. In *Proceedings of the 26th International Conference on World Wide Web*, pages 479–488, 2017.

[27] Yunhui Guo, Yandong Li, Liqiang Wang, and Tajana Rosing. Depthwise convolution is all you need for learning multiple visual domains. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 8368–8375, 2019.

[28] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pages 1737–1746. PMLR, 2015.

[29] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[30] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[31] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.

[32] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[33] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. Ai benchmark: Running deep neural networks on android smartphones. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, pages 0–0, 2018.

[34] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. Ai benchmark: All about deep learning on smartphones in 2019. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3617–3635. IEEE, 2019.

[35] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[36] Gangwon Jo, Won Jong Jeon, Wookeun Jung, Gordon Taft, and Jaejin Lee. Opencl framework for arm processors with neon support. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, pages 33–40, 2014.

[37] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017.

[38] Dohyun Kim. A study of user data integrity during acquisition of android devices. 2013.

[39] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[40] Stefanos Laskaridis, Alexandros Kouris, and Nicholas D Lane. Adaptive inference through early-exit networks: Design, challenges and directions. *arXiv preprint arXiv:2106.05022*, 2021.

[41] Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2020.

[42] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495), 2020.

[43] Ilias Leontiadis, Stefanos Laskaridis, Stylianos I Venieris, and Nicholas D Lane. It's always personal: Using early exits for efficient on-device cnn personalisation. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 15–21, 2021.

[44] Yuhang Li, Wei Wang, Haoli Bai, Ruihao Gong, Xin Dong, and Fengwei Yu. Efficient bitwidth search for practical mixed precision neural network. *arXiv preprint arXiv:2003.07577*, 2020.

[45] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 389–400, 2018.

[46] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[47] Chunjie Luo, Xiwen He, Jianfeng Zhan, Lei Wang, Wanling Gao, and Jiahui Dai. Comparison and benchmarking of ai models and frameworks on mobile devices. *arXiv preprint arXiv:2005.05085*, 2020.

[48] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019.

[49] Felix Mues. Optimization of opengl streaming in distributed embedded systems. 2020.

[50] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[51] Vineel Pratap, Qiantong Xu, Jacob Kahn, Gilad Avidov, Tatiana Likhomanenko, Awni Hannun, Vitaliy Liptchinsky, Gabriel Synnaeve, and Ronan Collobert. Scaling up online speech recognition using convnets. *arXiv preprint arXiv:2001.09727*, 2020.

[52] Xiuquan Qiao, Pei Ren, Schahram Dustdar, and Junliang Chen. A new era for web ar with mobile edge computing. *IEEE Internet Computing*, 22(4):46–55, 2018.

[53] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[54] Pei Ren, Xiuquan Qiao, Yakun Huang, Ling Liu, Calton Pu, and Schahram Dustdar. Fine-grained elastic partitioning for distributed dnn towards mobile web ar services in the 5g era. *IEEE Transactions on Services Computing*, 2021.

[55] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[56] Graham Sellers and John Kessenich. *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional, 2016.

[57] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[58] Saurabh Singh and Shankar Krishnan. Filter response normalization layer: Eliminating batch dependence in the training of deep neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11237–11246, 2020.

[59] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[60] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[61] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[62] Xiaohu Tang, Shihao Han, Li Lyna Zhang, Ting Cao, and Yunxin Liu. To bridge neural network design and real-world performance: A behaviour study for neural networks. *Proceedings of Machine Learning and Systems*, 3, 2021.

[63] Robert Tornai and Péter Fürjes-Benke. Compute shader in image processing development. 2021.

[64] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. 2011.

[65] Haoyu Wang, Hao Li, and Yao Guo. Understanding the evolution of mobile app ecosystems: A longitudinal measurement study of google play. In *The World Wide Web Conference*, pages 1988–1999, 2019.

[66] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus. In *MobiCom*, pages 215–228, 2021.

[67] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. Understanding and detecting fragmentation-induced compatibility issues for android apps. *IEEE Trans. Software Eng.*, 46(11):1176–1199, 2020.

[68] Hao Wu, Jinghao Feng, Xuejin Tian, Edward Sun, Yunxin Liu, Bo Dong, Fengyuan Xu, and Sheng Zhong. Emo: Real-time emotion recognition from single-eye images for resource-constrained eyewear devices. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 448–461, 2020.

[69] Ruofan Wu, Feng Zhang, Zhen Zheng, Xiaoyong Du, and Xipeng Shen. Exploring deep reuse in winograd cnn inference. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 483–484, 2021.

[70] Liang Xu, Xuanwei Zhang, and Qianqian Dong. Cluecorpus2020: A large-scale chinese corpus for pre-training language model. *arXiv preprint arXiv:2003.01355*, 2020.

[71] Mengwei Xu, Zhe Fu, Xiao Ma, Li Zhang, Yanan Li, Feng Qian, Shangguang Wang, Ke Li, Jingyu Yang, and Xuanzhe Liu. From cloud to edge: a first look at public edge platforms. In Dave Levin, Alan Mislove, Johanna Amann, and Matthew Luckie, editors, *IMC '21: ACM Internet Measurement Conference, Virtual Event, USA, November 2-4, 2021*, pages 37–53. ACM, 2021.

[72] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. A first look at deep learning apps on smartphones. In *The World Wide Web Conference*, pages 2125–2136, 2019.

[73] Mengwei Xu, Feng Qian, Qiaozhu Mei, Kang Huang, and Xuanzhe Liu. Deeptype: On-device deep learning for input personalization service with minimal privacy concern. *IMWUT*, 2(4):1–26, 2018.

[74] Mengwei Xu, Feng Qian, Mengze Zhu, Feifan Huang, Saumay Pushp, and Xuanzhe Liu. Deepwear: Adaptive local offloading for on-wearable deep learning. *IEEE Trans. Mob. Comput.*, 19(2):314–330, 2020.

[75] Mengwei Xu, Tiantu Xu, Yunxin Liu, and Felix Xiaozhu Lin. Video analytics with zero-streaming cameras. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 459–472. USENIX Association, 2021.

[76] Mengwei Xu, Xiwen Zhang, Yunxin Liu, Gang Huang, Xuanzhe Liu, and Felix Xiaozhu Lin. Approximate query service on autonomous iot cameras. In Eyal de Lara, Iqbal Mohomed, Jason Nieh, and Elizabeth M. Belding, editors, *MobiSys '20: The 18th Annual International Conference on Mobile Systems, Applications, and Services, Toronto, Ontario, Canada, June 15-19, 2020*, pages 191–205. ACM, 2020.

[77] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 129–144, 2018.

[78] Jingyun Yang, Hengjun Wang, and Kexiang Guo. A subsequent words recommendation scheme for chinese input method based on deep reinforcement learning. In *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*, pages 1482–1487. IEEE, 2020.

[79] Hyunho Yeo, Chan Ju Chong, Youngmok Jung, Juncheol Ye, and Dongsu Han. Nemo: enabling neural-enhanced video streaming on commodity mobile devices. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[80] Yunhua Yin, Huifang Li, and Wei Fu. Faster-yolo: An accurate and faster object detection method. *Digital Signal Processing*, 102:102756, 2020.

[81] Salih Can Yurtkulu, Yusuf Hüseyin Şahin, and Gozde Unal. Semantic segmentation with extended deeplabv3 architecture. In *2019 27th Signal Processing and Communications Applications Conference (SIU)*, pages 1–4. IEEE, 2019.

[82] Jinrui Zhang, Deyu Zhang, Xiaohui Xu, Fucheng Jia, Yunxin Liu, Xuanzhe Liu, Ju Ren, and Yaoxue Zhang. Mobipose: Real-time multi-person pose estimation on mobile devices. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 136–149, 2020.

[83] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 863–879, 2020.

[84] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.