A Comprehensive Deep Learning Library Benchmark and Optimal Library Selection

Qiyang Zhang, Xiangying Che, Yijie Chen, Xiao Ma, *Member, IEEE,*, Mengwei Xu, *Member, IEEE,* Schahram Dustdar, *Fellow, IEEE*, Xuanzhe Liu, *Member, IEEE,* and Shangguang Wang, *Senior Member, IEEE*

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 libraries and provides quantitative results on their performance. In this paper, we first build a comprehensive benchmark that includes 6 representative DL libraries and 15 diversified DL models. Then we perform extensive experiments on 10 mobile devices, and the results reveal the current landscape of mobile DL libraries. For example, we find that the best-performing DL library is severely fragmented across different models and hardware, and the gap between DL libraries can be rather huge. In fact, the impacts of DL libraries can overwhelm the optimizations from algorithms or hardware, e.g., model quantization and GPU/DSP-based heterogeneous computing. Motivated by the fragmented performance of DL libraries across models and hardware, we propose an effective DL Library selection framework to obtain the optimal library on a new dataset that has been created. We evaluate the DL Library selection algorithm, and the results show that the framework at it can improve the prediction accuracy by about 10% than benchmark approaches on average.

Index Terms—Benchmark, Deep Learning, Mobile Devices, Library Selection.

1 Introduction

Deep learning (DL) has become an indispensable functional module for today's smartphones, widely adopted in applications like input method, AR/VR, voice assistant, etc [1], [2]. A noteworthy trend is that more and more DL 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. 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 [3], [4].

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 [5]–[7]. Consequently, optimizing the inference performance has been the theme of both academia [1], [3], [8] and industry [9], [10] 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 libraries*) underexplored. These libraries share the same goal: executing the inference task solely on smartphones. 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 [11]. Furthermore, due to the severely fragmented ecosystem of smartphones [12], there exists a mass of heterogeneous DL library alternatives for app developers [3], [4], making it difficult and labor-intensive to compare their suitability into specific models.

To gain in-depth understandings of the performance of modern DL libraries, we first build a comprehensive benchmark for on-device DL inference, namely `MDLBench`. The benchmark includes 6 popular, representative DL libraries on mobile devices, i.e., `TFLite`, `PyTorchMobile`, `ncnn`, `MNN`, `Mace`, and `SNPE` [13]–[18]. It contains 6 DL models compatible with all above DL libraries, 8 models compatible with at least 3 above DL libraries, and 1 model compatible with 2 DL libraries, spanning from image classification, object detection, to NLP. Compared to existing AI benchmarks, our benchmark triumphs at the aspect of rich support for various DL libraries and models. In addition to the completeness, we also instrument the DL libraries 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 libraries and therefore provide more insightful implications to developers.

Based on our benchmark, we perform extensive experiments to demystify the performance of DL libraries 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 as follows:

(1) The performance of the 6 DL libraries benchmarked is severely fragmented across different models and hardware (§3.1). There is no **One-Size-Fits-All** DL library that performs best on all scenarios (model⊗device), yet each DL library has at least one best-performing scenario. Even for the same model, there are different DL libraries that perform the best on different devices.

(2) The impacts of DL software may overwhelm the

xxxx-xxxx/0x/$xx.00 2023 IEEE Published by the IEEE Computer Society

model/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.

(3) Cold-start inference of DL libs is significantly slower than warm inference (§3.4). 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.

(4) During the evolution of DL libraries, performance bugs are introduced for many times (§3.5). By benchmarking the weekly version of `TFLite` and `ncnn` since early 2018, we find that the overall performance of DL libraries is improving yet becomes relatively stable since 2020.

Among the above observations, the severely fragmented inference performance of libraries across different models and hardware is a critical but unexplored issue. In practical applications, developers usually use one library to run different models [4]. When the models do not fit the libraries accurately, the inference performance will be significantly degraded and even user experience will be compromised. For example, as one of the most commonly used models, vgg16 in Tab. 3 shows a 54.3× inference time gap between the best and the worst libraries. Moreover, one app usually integrates more than one library (as one app is usually developed by different engineers, who introduce different libraries), leaving the space for improving the inference performance by selecting the most proper library for each model of an app. No prior work has dived deep into the inference-time oriented library selection issue for models. In this paper, we seek to address this issue, aiming at optimizing the inference time of models.

Selecting the most optimal libraries for different models faces a key challenge. To obtain the optimal library for models, the inference time of different libraries should be obtained. Yet measuring the inference time on real-world apps is costly, or even infeasible due to the high inference time overhead, especially for the worst-performing library. To deal with this challenge, we propose a prediction-based library selection framework to select the most proper library for each model with low time overhead. The library selection framework can train a prediction model to select the optimal library directly based on the characteristics of each model, instead of selecting based on the inference performance after substantial measurements. However, the prediction-based library selection framework requires a dataset recording the inference performance of running the same models on different libraries. However, there is no off-the-shelf dataset that can be used directly. Even `MDLBench` only provides fewer same models on the libraries. To address this concern, we create a dataset that contains 1127 state-of-the-art models with 13 operator types and configurations. These models can run on 5 popular DL libraries, i.e., `TFLite`, `ncnn`, `MNN`, `Mace`, and `PyTorchMobile` [13], [15]–[18]. For fairness, this dataset also ensures that the same models can be generated from different libraries. We perform extensive experiments based on the dataset by

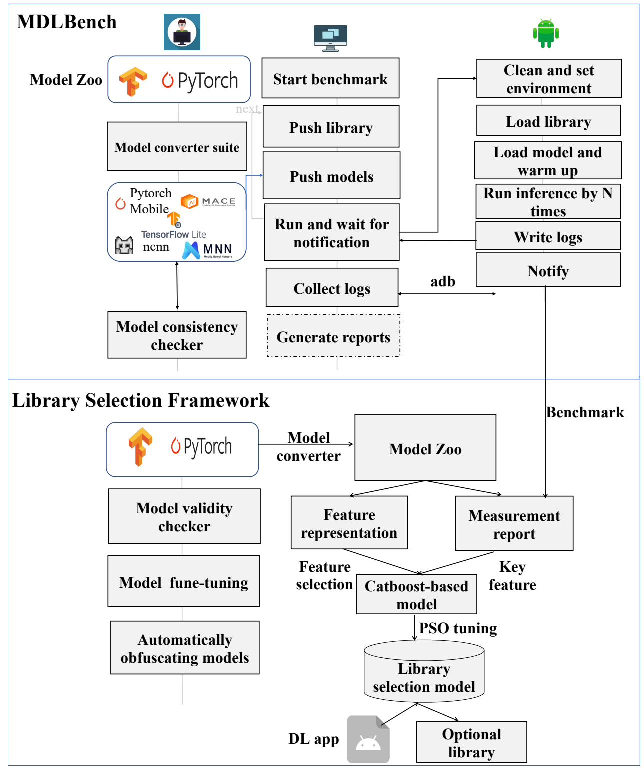


Fig. 1. The overall architecture of our work includes two tiers: the first tier shows the workflow of `MDLBench` and the second tier shows the library selection framework.

`MDLBench` automatically and obtain the inference time of models offline, which are the basis of the library selection model in this framework.

Our main contributions are as follows.

- We design and implement `MDLBench`, a fully automatic, comprehensive benchmark for DL libraries. The full benchmark suite and measurement results used in this work are available [1].
- We conduct extensive experiments with `MDLBench` on diverse hardware devices and models. For the first time, the results reveal a complete landscape of the current DL library ecosystem. We also summarize the insightful observations and practical implications.
- We propose a prediction based library selection framework for models to derive the most proper library with low time overhead. To enable the prediction based library selection framework, a new dataset that contains 1,127 models with 13 operator types and configurations has been created.
- We implement and evaluate the proposed framework for models, and demonstrate that it can improve the prediction accuracy by about 10% than benchmark approaches on average.

2 Benchmark & Methodology

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

• **Rich support** Tab. 1 summarizes the DL libraries (6 in total), models (15 in total), and hardware processor

1. https://github.com/UbiquitousLearning/

(CPU/GPU/DSP) `MDLBench` currently supports. Being able to test many DL libraries under various contexts is critical to obtain a complete landscape of the DL library ecosystem, because the performance optimization is quite ad-hoc to models and hardware. Among the large amount of DL libraries available for developers, we select 6 most representative ones from a "market" perspective. We follow the prior works [4] to detect the DL libraries 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 libraries 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 library while all above are not. The models we use come from two sources. One is the model zoo of TensorFlow and PyTorch [19], [20]. The other is by using the built-in converters of each DL library to convert models to different formats [15]–[18]. `MDLBench` also incorporates a module to automatically check the equivalence of the same model generated for different DL libraries.

Workflow The first tier of Fig. 1 shows the overall workflow of `MDLBench`. For each testing, the desktop-side benchmark iterates over each DL library. It first pushes the library and corresponding models generated as aforementioned to the devices through adb [36]. 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 [37], [38], we always use 4 big cores to run the DL libraries 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 Tab. 2 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.

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 libraries, and how the inter-play with the impacts from algorithm and hardware.

3 PERFORMANCE ANALYSIS AND IMPLICATIONS

This section presents our analysis of DL libraries for smartphone. 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. Finally, we show implications to different roles in the mobile DL ecosystem.

3.1 Performance Fragmentation

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

(1) **There is no one-size-fit-all DL library for optimal performance across models and hardware.**

Each DL library has at least one best-performing scenario, except that `PyTorchMobile` does not support GPU acceleration. Even for the same model, its corresponding best-performing DL library may change across different hardware. For instance, the best-performing DL libraries 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 [39] 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 library developers need to handcraft 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 library optimized for each scenario.

(2) **The performance gap of DL libraries can be large.**

The "gap" is defined as the ratio of inference time of two DL libraries (the longer one divided by the shorter one). The numbers in parentheses are average values. Surprisingly, though those DL libraries 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 libraries 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×.

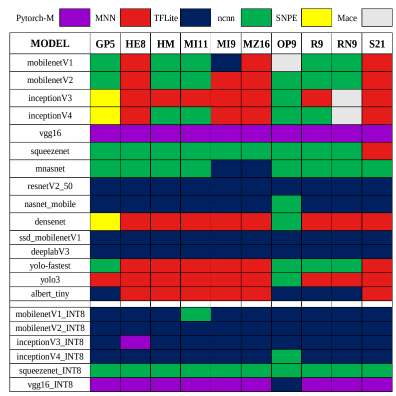
(3) **GPU backend choices further exaggerate the fragmentation.** Even on the same GPU, there are different backend choices implemented by DL libraries. For example, `MNN` implements three backends: Vulkan, OpenGL and OpenCL [40]–[42]. Interestingly, as shown in Fig. 2(b), different GPU backend choices 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

TABLE 1
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. "C", "SS", "OD", and "TC" represent "image classification", "semantic segmentation", "object detection", and "text classification", respectively.

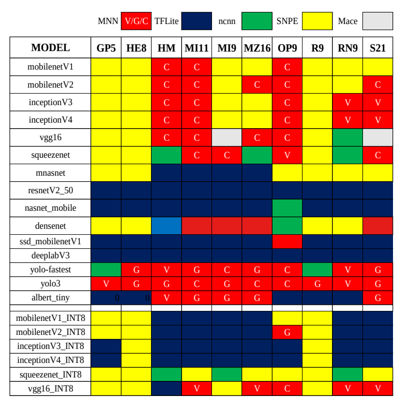
Models	Tasks	TFLite	ncnn	mnn	MACE	PyTorchMobile	SNPE
mobilenetV1 [21]	C	$C_{32,8}$-$G_{32,8}$-D_8	$C_{32,8}$-$G_{32,8}$	$C_{32,8}$-$G_{32,8}$	$C_{32,8}$-G_{32}	$C_{32,8}$	$C_{32,8}$-$G_{32,8}$-D_8
mobilenetV2 [22]	C	$C_{32,8}$-$G_{32,8}$-D_8	$C_{32,8}$-$G_{32,8}$	$C_{32,8}$-$G_{32,8}$	$C_{32,8}$-G_{32}	$C_{32,8}$	$C_{32,8}$-$G_{32,8}$-D_8
inceptionV3 [23]	C	$C_{32,8}$-$G_{32,8}$-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 [24]	C	$C_{32,8}$-$G_{32,8}$-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 [25]	C	$C_{32,8}$-$G_{32,8}$-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 [26]	C	$C_{32,8}$-$G_{32,8}$-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 [27]	C	C_{32}-G_{32}	-	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}	-
densenet [28]	C	C_{32}-G_{32}	-	C_{32}-G_{32}	-	C_{32}	C_{32}-G_{32}
mnasnet [29]	C	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}	C_{32}-G_{32}
resnetv2_50 [30]	C	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}	C_{32}-G_{32}
deeplabv3 [31]	SS	C_{32}-G_{32}	-	C_{32}-G_{32}	C_{32}-G_{32}	-	-
ssd_mobilenetV1 [32]	OD	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}	-
yolo-fastest [33]	OD	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	-	-	-
yolo3 [34]	OD	C_{32}-G_{32}	C_{32}-G_{32}	C_{32}-G_{32}	-	-	-
albert_tiny [35]	TC	C_{32}-G_{32}	-	C_{32}-G_{32}	-	-	-

TABLE 2
The tested devices and their specifications.

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 C2	4GB
Redmi Note9 Pro	RN9	Snapdragon 720G	Adreno 618	6GB
Samsung S21	S21	Snapdragon 888	Adreno 660	8GB



(a) CPU.



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

Fig. 2. The best-performing DL library (smallest inference time) with different models and devices.

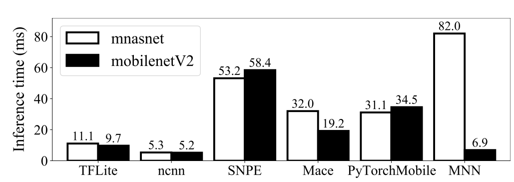


Fig. 3. The average inference time of mobilenetV2 and mnasnet with different libraries on MI11.

interfaces highly optimized for mobile devices [41]. 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 libraries. As shown in Fig. 3, squeezenet runs faster than mobilenetV2 with `SNPE`, `PyTorchMobile`, while mobilenetV2 runs faster with other DL libraries. The reason of such behavior can be these libraries adapt to a wide variety of operators and the operators are implemented in different ways. The same operator even has different latency because the same operator has fewer implementations on

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3301973

TABLE 3
The performance gaps of different DL libraries.

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

one library. For instance, convolution operators employ different algorithms depending on the hyperparameters, such as Winograd for 3×3 and direct convolution for 5×5 convolution [43].

†**Summary** *The best-performing DL library is highly fragmented across models and hardware. Such fragmentation may even overwhelm the model designs and hardware capacity improvement. To pursue optimal performance in a mobile DL app, the developers need to incorporate different DL libraries 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 libraries.*

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. 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. Fig. 4 summarizes the best inference performance across DL libraries 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 [42]). 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 libraries 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 libraries. (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.

†**Summary** *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 Fig. 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 Fig. 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 old ones, e.g., Snapdragon 870 vs. 855 on `TFLite`, even though 870 is equipped with stronger CPU and faster memory access speed. This is mainly because Snapdragon 855 is a more popular SoC for which the DL libraries 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 [44]. Secondly, the DL libraries 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].

†**Summary** *Our findings motivate DL library developers to focus on GPU-side optimization [37], 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. Fig. 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 [45], which are integrated with Hexagon vector extension (HVX) acceleration. Meanwhile, the Winograd algorithm is used to accel-

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

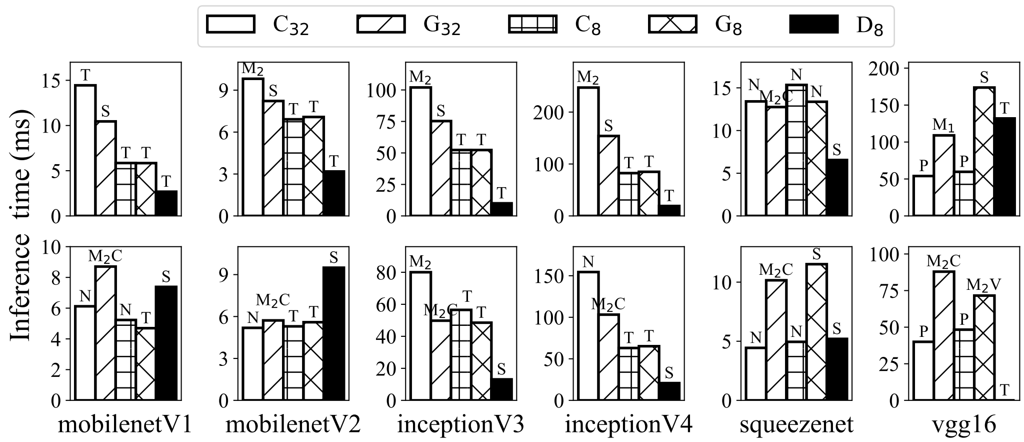


Fig. 4. The optimal inference speed among all deep learning libraries for RN9 and MI11. "T", "N", "S", "P", "M_1", and "M_2" are short for `TFLite`, `ncnn`, `SNPE`, `PyTorchMobile`, `Mace`, and `MNN` respectively as the best-performing DL libraries. "V", "C", "G" indicate different GPU backends. "C/G/D": mobile CPU/GPU/DSP. The subscripted 32 and 8 represent different model precision, i.e., float32 and int8, respectively. The height of the bar represents the inference time of best-performing library. Note that, for DSP, we leave out the performance of vgg16 with SNPE since the model does not work with DSP on MI11.

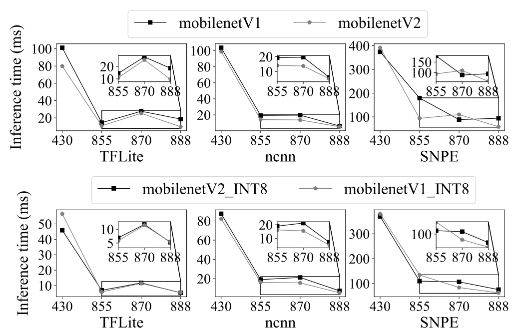


Fig. 5. The performance across different SoCs.

erate 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 libraries are not currently well tuned for.

†**Summary** *In most cases, more powerful CPUs and accelerators (GPU and especially DSP) can speed up the model inference. However, there are cases that DL libraries perform even worse on those hardware. In other words, the current DL libraries can not fully exploit the capacity of each hardware. Our findings motivate DL library developers to focus on optimization on heterogeneous processors, 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 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, "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. Fig. 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 [46] and Winograd [47] implementation of convolution operation require to transform the original convolution kernel matrix to a different shape.

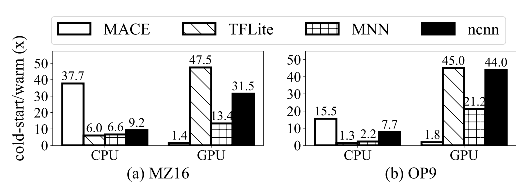


Fig. 6. The ratio of cold-start inference to warm inference. Numbers are averaged across all DL models.

Fig. 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

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3301973

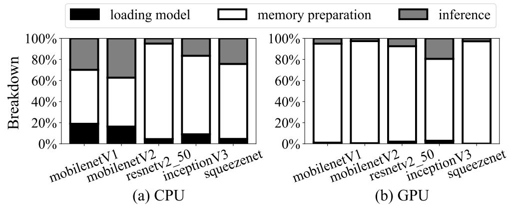


Fig. 7. The breakdown of cold-start inference time.

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 [48].

†**Summary** *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.*

3.5 Longitudinal Analysis

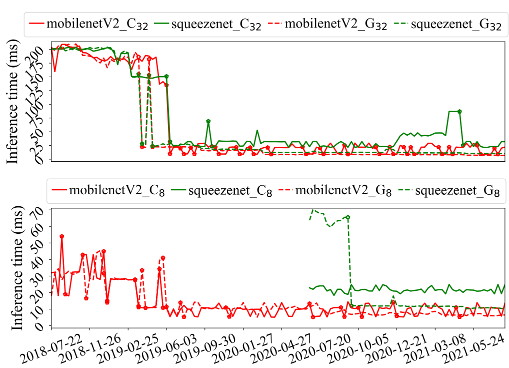


Fig. 8. The inference performance evolvement across time of TFLite tested on HM device.

We then longitudinally analyze how the performance of DL libraries evolves across time. We select 2 DL libraries 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 libraries are continuously improving in early years, but becomes relatively stable since 2020. As shown in Fig. 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 libraries is relatively stable and there are very few nontrivial improvements. It indicates that the DL library 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 squeezenet. The reason of such "partial improvement" is the same as the fragmentation of DL libraries as mentioned in §3.1.

†**Summary** *The current open-source ecosystems of DL libraries 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 libraries 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.*

3.6 Implication

From the above analysis, our findings paint a promising picture of DL library, motivating future research and development. In this section, we discuss actionable implications to different roles in the mobile DL ecosystem as follows:

- **For DL app/model developers** (1) It is extremely challenging in selecting a proper DL library due to the severe fragmentation. To pursue optimal performance under each scenario, they have to embed different DL libraries into the apps and load one dynamically based on the model and hardware settings. (2) A more lightweight model (fewer FLOPs) does not always run fast. The impacts from software at deployment needs to be considered during the model designing.
- **For DL library engineers and researchers** (1) It is time to review the pros and cons of different DL libraries 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. Tools that can automatically identify such bugs timely, either through dynamic or static analysis, are urgently needed. (2) 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. (3) Performance bugs bring negative impacts to the open-source ecosystem of DL libraries but are difficult to be fully eliminated due to the aforementioned fragmentation.

4 Deep Learning Library Selection Framework

Developers always use these libraries designed and optimized for smartphones to build their DL apps. A suitable library for models of apps is specially selected for the smallest inference time, based on the observation that one app

TABLE 4
The proportion of operators covered in the dataset.

Operator	Example	Percentage in the dataset
Conv	Conv2D DepthwiseConv2D	50.4%
Pooling	MaxPool AvgPool	42.3%
Activation	Softmax Relu Prelu	100%
Linear	Mul BiasAdd	100%
Other	Add Sub Reshape Dropout	99.5%

may incorporate multiple libraries [4]. However, selecting the optimal library for models is a very challenging task, as measuring the inference time of running models on the integrated libraries is extremely costly. So we are motivated to propose an efficient library selection framework to predict an optimal library for models.

The second tier of Fig. 1 shows the library selection framework in detail. We first create a large-scope dataset including existing common multiple models, as `MDLBench` only provides a small number of models due to the unsupported operators in model conversion between libraries. Based on the collected model zoo, we utilize `MDLBench` to obtain the inference reports of running multiple models on different libraries. We consider the library with the smallest inference time reported by each model as the optimal library. We collect inference reports to make a new library selection-oriented dataset, which is used as the foundation for library selection. We propose a library selection framework by training a CatBoost-based selection model and further improving its performance by tuning the parameters.

4.1 Dataset Creation

We first create a representative large-scale dataset, as there is no off-the-shelf large dataset that we can use directly. To this end, we design and implement a random model generator and consider the common models used in apps. Specifically, we randomly transform model structure and output by automatically obfuscating models to generate Tensorflow [49] and Pytorch [50] format considering the common models (i.e., mobilenet [21], vgg [25], and MLP [10]) and their variants. As shown in Tab. 4, we also consider the models consisting of any primitive operator type and the various edge connections between operators. In total, our dataset contains 1,127 models with 13 types of operators. We also ensure the richness and validity of our dataset. Fig. 9 shows the probability density distribution of parameters and FLOPs in the dataset. The FLOPs range from 900k to 11G and the parameters range from 400k to 30M. The results are consistent with the fact that more than 65% of the models in the industry fall into this above range [4]. In other words, the models in the dataset can be applied

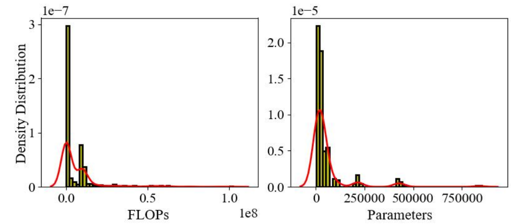


Fig. 9. The density distribution of the number of parameters and FLOPs in the models (i.e., the probability density distribution randomly generated for each model).

in real-world apps. The same model types in different DL libraries are also generated from the *Model converter suite* to ensure the equivalence of models from the dataset.

These models can run on 5 popular DL libraries, i.e., `TFLite`, `ncnn`, `MNN`, `Mace`, and `PyTorchMobile` [13], [15]–[18]. Compared to the benchmark, SNPE is not included due to the incompatibility. Although the workflow in the benchmark incorporates *Model converter suite*, an increase in the number of test models heightens the risk of conversion failure [51]. Additionally, SNPE's closed-source nature complicates its utilization. Its primary support for Snapdragon platforms further limits cross-platform deployment [18]. In order to maintain the reliability and validity of our experiments, we opted for libraries that are compatible with as many models as possible.

4.2 Improved Library Selection framework

As shown in the second tier of Fig. 1, we extract inference reports running models in the dataset on the libraries as the foundation of library selection. Note that feature extraction is low-cost, easy to identify, and low in information distortion. We assemble these key features, as outlined in Tab. 5, into unique vectors, including memory usage, computational demands, and model structure, leveraging a suitable feature representation method [52]. Memory usage includes the model parameters and generated intermediate results, such as feature maps. We employ FLOPs as the standard to measure computational complexity. Model structure encompasses the type and quantity of vital operators, such as Conv2D, DepConv, Mul, and BiasAdd.

Here we employ the Boosting-based algorithms for the library selection task, since the algorithms assign discrete variables to finite clusters and encode them in Oneshot. Compared with similar algorithms such as XGBoost [53], CatBoost automatically merges discrete features into internal ones and applies them to training models. CatBoost also overcomes the overfitting caused by the gradient bias of traditional Boosting training. Furthermore, the collected reports maybe have a severe imbalance due to the optimization differences among libraries. To address the issue, we consider a comprehensive loss including two aspects: misselection-based cross-entropy and performance error between prediction and ground truth.

Cross-entropy is always used in prediction for multi-classification tasks. Motivated by its advantages, misselection-based cross-entropy for library selection tasks

TABLE 5
The representation of key features influencing inference performance.

Item	Features	Explanation
Parameter Number and Distribution	Number	/
	Number of Vital OPs	Conv2D/DepConv/ Mul/BiasAdd
Computational Complexity	Number	FLOPs In Tatal
	Number of Each OP	/
Model Structure	OP Composition	Binary
	Number of Each OP	/
	Type	CNN/RNN/MLP

is summarized in Eq.(1).

$$L(y_i, f(x_i)) = \frac{1}{d}\sum_{j=1}^{d} w_j * y_j log_2 p_{i,j}, \tag{1}$$

where x_i represents a unique feature vector. y_i represents the actual optimal library of input x_i. d represents the number of libraries. $p_{i,j} = \frac{e^{f_j(x_i)}}{\sum_{i=1}^{d} e^{f_j(x_i)}} \in (0,1)$ represents the probability that the selection result of input x_i is the j library, where $f(x_i)$ is the optimal output obtained by x_i. Due to the large difference in the optimization on different DL libraries, there exists an unbalanced number in each library of the dataset. Therefore, we introduce the misselection cost as a penalty. For fairness, DL libraries with smaller number data ensure slightly larger weights by tuning different cost weights to DL libraries. With the misselection cost, where $w = [w_0, w_1, ...w_j]$ is the weight of the misselection cost.

Performance error L_{x_i,y_i} is used to evaluate the performance between the prediction and ground truth, which is defined in Eq.(2):

$$L_{x_i,y_i} = \frac{t_{x_i,y_i}}{t_{x_i,best}} + \lambda_0, \tag{2}$$

where λ_0 is the error penalty factor. t_{x_i,y_i} is the predicted optimal inference time. $t_{x_i,best}$ is the actual optimal inference time.

To address the issue of setting parameters caused by manual design and grid parameter search, we introduce Particle Swarm Optimization (PSO) to improve the performance of CatBoost algorithm. Obviously, selecting the suitable hyperparameters is a very challenging task since CatBoost has more than 20 hyperparameters such as the estimators, the learning rate, etc [54]. We exploit PSO to optimize the library selection algorithm, as CatBoost assumes that each selection has the same weight. It is difficult to set the weights of CatBoost in the case of unbalanced selection, so we dynamically obtain the global optimal selection to make balance the performance as possible. As shown in Tab. 3, due to the large performance error between the libraries, the sensitivity of misselection is related to the

Algorithm 1: PSO-W-CatBoost

input : population size of particle swarm N; algorithm iteration number I; all parameters of the models
output: the optimized selected library
Initialize the N particles;
Initialize pbest(t) and gbest(t) of all particles;
Train catBoost and compute $F(t)$ according to Eq.(3);
while $t < \tau$ **do**
  foreach *particle of total N particles* **do**
    update velocity of each particle according to Eq.(4);
    update the position of each particle according to Eq.(5);
    Recompute F(t+1);
    while *F(t+1) < F(t)* **do**
      F(t)=F(t+1)
  update pbest(t+1);
  update gbest(t+1);
  t←t+1;
Train again CatBoost model with W_{best} obtained by PSO;
Return the optimized selected library;

performance error. For instance, if the misselection inference time is lower than the average one, misselection may seriously weaken user experience. We accept the results when the actual selection performance is close to the best. Therefore, we employ performance loss as the evaluation of library selection task.

The fitness function reflects the loss of individual extremum to the library selection tasks. The larger the fitness is, the smaller the loss is; vice versa. The testing accuracy can directly reflect the performance of the selection algorithm. Simultaneously, the performance loss is used to evaluate the misselection. Therefore, fitness function $F(t)$ includes the inverse of the performance loss and the testing accuracy at time t, as shown in Eq.(3).

$$F(t) = \lambda_1 \frac{M}{\sum_{i=1}^{M} L_{x_i,y_i}} + \lambda_2 R_{acc}, \tag{3}$$

where λ_1 and λ_2 are weights, $R_{acc} = \frac{\sum_{i=1}^{M} \mathbb{I}(y_i = f_{(x_i)})}{M}$ is testing accuracy, $\mathbb{I}(y_i = f_{(x_i)})$ equals to 1 only when the chosen library is the best. M is the testing number.

Here we regard key parameters (e.g., the depth of decision tree d, learning rate lr, the penalty factor e, etc) as the particles of PSO. In the iteration and weight tuning, the position and velocity of each particle should be calculated and adjusted according to the individual extremum and global optimal solution. In each update, the magnitude and direction of the velocity are updated according to the gap between the local and the global position, and the local position is also updated according to the direction change of the velocity. The velocity and position updates are shown in Eq.(4) and Eq.(5), where t represents the time t, $pbest$ and $gbest$ represent the local optimal and global optimal position respectively, a_1 and a_2 represent random factors, w_1, w_2 and w_3 represent the current velocity, local updating factor and global updating factor optimization coefficient respectively. $p_i(t)$ is the position of particle i at time t.

$$v_{i+1}(t+1) = w_i v_i(t) + w_2 a_1 (pbest_i(t) - p_i(t)) + w_3 a_2 (gbest - p_i(t)), \quad (4)$$

$$p_{i+1}(t+1) = p_i(t) + v_{i+1}(t+1). \quad (5)$$

The pseudocode of the improved PSO-W-CatBoost-based algorithm is shown in algorithm 1. The input of the algorithm is based on the model feature representation vector extracted by feature engineering, and the output is the optimized selected library. All parameters are initialized, and the optimal combination of parameters and weights is learned until $t<\tau$. The algorithm establishes the CatBoost predictor based on the above training.

5 Evaluation

In our experiment, we train high-accuracy library selection models to obtain the optimal library on the built dataset, in which the ratio of the training set to the test set is 7:3. The complex unstructured model is converted to a unique feature vector through feature construction and extraction. We choose GP5 as the tested device from Tab. 2 to carry out experiments. In the following, we further introduce benchmark algorithms, evaluation metrics and discuss the experimental results.

5.1 Benchmark algorithms

To evaluate the performance of the PSO-W-CatBoost algorithm, three algorithms used in similar tasks are introduced as benchmarks, which are listed as follows.

- Hierarchical Support Vector Machine [53] (labeled as SVM) obtains the library selection by training the SVM classifier based on the nodes in the decision tree.
- Extreme Gradient Boosting [54] (labeled as $XGBoost$) minimizes the fitness function to obtain the library selection based on the generation and pruning of decision trees.
- Recurrent Graph Convolutional Network [55] (labeled as $RGCN$) obtains the library selection by the encoding/decoding of feature vector and graph relationship. $RGCN$ also converts high-dimensional graph relationships into feature vectors and obtains the graph feature from the feature vectors.

5.2 Metrics

We use the following metrics commonly used in classification tasks to evaluate library selection tasks, as the two tasks have similar targets. For the selection task, the Macro method [56] is used to evaluate various accuracy of algorithms. The overall accuracy is represented by the average accuracy of each selection.

- $accuracy$ reflects the proportion of correctly selected libraries in all libraries. It's the simplest and most intuitive metric for the library selection task.
- $precision$ directly reflects the proportion of the correctly selected libraries in selected libraries.
- $recall$ reflects the proportion of the selected libraries in ground truth libraries.
- F_{score} is a comprehensive metric considering $Precision$ and $Recall$ simultaneously. It is a harmonic average of $precision$ and $recall$.

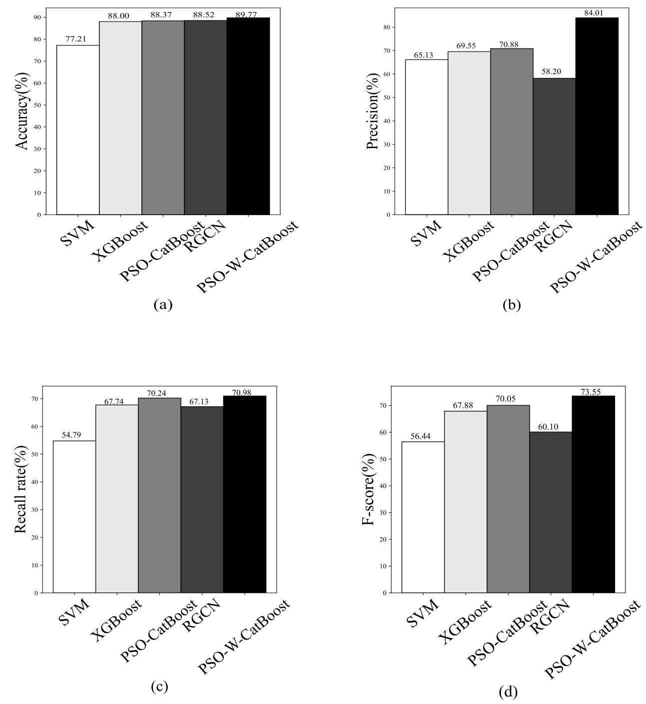


Fig. 10. Comparison of Prediction Accuracy of Different Algorithms.

5.3 Experimental Results

To obtain a high-accuracy library selection and evaluate the performance of the library selection framework, Fig. 10 shows the detailed selection results on the $accuracy$, $precision$, $recall$, and F_{score}. To sum up, the proposed framework can improve the prediction accuracy by about 10% than benchmark approaches on average. From the perspective of service providers, although the algorithm does not improve much compared to other benchmarks, high-accuracy service provision not only provides users with high-quality services but also generates great benefits. As shown in Fig. 10, the performance of SVM and $XGboost$ is not as good as that of the $PSO-W-CatBoost$ algorithm. It is difficult for SVM to find a suitable kernel function because conventional SVM can only solve the binary classification problem. $XGboost$ is better at dealing with low-dimensional feature data and cannot deal with high-dimensional data. Besides, we also deem that a similar observation is also found on other smartphones.

As shown in Fig. 10(a), the $PSO-W-CatBoost$ algorithm has the highest $accuracy$. The performance of $PSO-W-CatBoost$ is better than that of $PSO-CatBoost$, among which $accuracy$ is improved by about 1.4%. That's because $PSO-W-CatBoost$ focuses on weight tuning. For example, the size N of PSO is set to 20, and the optimal of $PSO-W-CatBoost$ is $w_{best} = \{d =$

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3301973

$8, lr = 0.38, e = [0.12, 0.15, 0.60, 0.27, 0.63]\}$. It is worth noting that the *accuracy* of RGCN is not lowered much than that of $PSO-CatBoost$. However, it is lower than that of $PSO - CatBoost$ and $PSO - W - CatBoost$ in terms of *precision*, as shown in Fig. 10(b). It is also consistent with the F_{score} in fig. 10(d) as F_{score} is a harmonic average of *precision* and *recall*. In addition, the distribution of the dataset is also not uniform. We checked and observed that there are smaller samples in some libraries such as `Mace`. It can be concluded that the $RGCN$ algorithm is dependent on the library with a large number of models.

5.4 Analysis of Convergence

To verify the effectiveness of the weighted cost, we make a comparison with $CatBoost$. It is worth noting that, $CatBoost$ abandons the tuning of classification cost compared with $PSO - W - CatBoost$. Fig. 11 shows the performance error loss change of $PSO-W-CatBoost$ and $CatBoost$ with the training iteration, where the red line represents the convergence number for $PSO - W - CatBoost$ while the green represents CatBoost. The result explicitly shows that $PSO - W - CatBoost$ converges faster with a smaller iteration number than $CatBoost$.

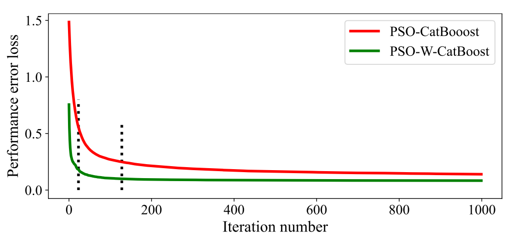


Fig. 11. The performance error loss change of $PSO - W - CatBoost$ and $CatBoost$.

6 Related Works

Mobile DL In recent years, there is a notable trend to move DL inference into local devices instead of offloading to remote servers [1], [8], [57], [58]. 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 [49], [50], [59]. There have been some optimization research efforts addressed to reduce the overhead of DL on smartphones, e.g., offloading, model quantization and sparsity [60]–[62]. 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 libraries. This work is motivated by many years of efforts at this lane.

AI benchmarks There exist a few AI benchmarks for diversified scenarios, e.g., datacenter servers or edges, inference or training, etc [63]. This work explicitly targeted at inference on mobile devices. Besides, the ecosystem of on-device DL libraries is much more fragmented than servers due to the high fragmentation of mobile hardware. Furthermore, a number of studies focus on DL libraries analysis. Amin et al. [64] compared only the `TFLite` and `PyTorchMobile` in the terms of robustness and adversarial attacks. Consequently, the results are limited in small-scale project from a specific perspective. Luo et al. [65] proposed the benchmark suite for evaluating the abilities of mobile devices across different libraries. MLPerf [66] proposed high-level rules for more flexible benchmark of the libraries. Tang et al. [67] 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 libraries across heterogeneous smartphones. Compared to similar benchmarks focusing on DL libraries, `MDLBench` has richer support for various DL libraries and models.

Empirical study of mobile DL One line of studies mainly focus on DL apps/systems/models. Xu et al. [4] demystified how smartphone apps exploit DL models by deeply analyzing Android apps. Wang et al. [68] made efforts towards the evolution of mobile app ecosystem. Andrey et al. [69] 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 libraries.

Library selection is a key but unexplored topic. There have been related works in different algorithms in service selection. For instance, Pascal et al. [70] discussed selection based on contextual scenarios, such as algorithm configuration and scheduling. It also provides an overview of the relevant selection algorithms for discrete and continuous problems. Sebastian et al. [71] proposed a common approach to model evaluation and selection. Basar et al. [72] focused on real-time machine vision applications running on resource-constrained embedded systems and proposed an adaptive model selection framework to reduce the impact of system contention. It is worth noting that these works do not focus on the performances of DL libraries across models and hardware. As a result, an orthogonal way to guarantee better service is to select the optimal library. Our work goes deep into the modern DL library ecosystem, providing the most suitable DL libraries for models in apps, thus greatly improving the utilization efficiency of DL libraries.

7 Conclusion and Future Work

In this work, we built the first comprehensive benchmark for DL libraries and conducted extensive measurements to quantitatively understand their performance. The results help reveal a complete landscape of the DL libraries ecosystem. Atop the observations, we summarize strong implications that can be useful to developers and researchers. Based on these findings, we propose the DL library selection algorithm to guarantee better service.

In the future, we will focus on the following three potential directions along this line: (1) We will try to maintain the platform based on the proposed benchmark suite to test and analyze the measurement results; (2) We will further open the measurement results to make it work properly for service provision.

Acknowledgment

A preliminary version of this paper appears as a conference paper in proceedings of the 31st Annual International World Wide Web Conference (WWW) 2022 [63].

References

[1] M. Xu, F. Qian, Q. Mei, K. Huang, and X. Liu, "Deeptype: On-device deep learning for input personalization service with minimal privacy concern," *IMWUT*, vol. 2, no. 4, pp. 1–26, 2018.

[2] S. Manoharan and P. Natu, "Development of a framework for a collaborative and personalised voice assistant," *Electronic Government, an International Journal*, vol. 17, no. 1, pp. 96–104, 2021.

[3] M. Almeida, S. Laskaridis, A. Mehrotra, L. Dudziak, I. Leontiadis, and N. D. Lane, "Smart at what cost? characterising mobile deep neural networks in the wild," *arXiv preprint arXiv:2109.13963*, 2021.

[4] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "A first look at deep learning apps on smartphones," in *The World Wide Web Conference*, pp. 2125–2136, 2019.

[5] M. Almeida, S. Laskaridis, I. Leontiadis, S. I. Venieris, and N. 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*, pp. 1–6, 2019.

[6] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[7] V. Pratap, Q. Xu, J. Kahn, G. Avidov, T. Likhomanenko, A. Hannun, V. Liptchinsky, G. Synnaeve, and R. Collobert, "Scaling up online speech recognition using convnets," *arXiv preprint arXiv:2001.09727*, 2020.

[8] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, "Deepcache: Principled cache for mobile deep vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pp. 129–144, 2018.

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

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

[11] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the top: What will drive computer performance after moore's law?," *Science*, vol. 368, no. 6495, 2020.

[12] L. Wei, Y. Liu, S. Cheung, H. Huang, X. Lu, and X. Liu, "Understanding and detecting fragmentation-induced compatibility issues for android apps," *IEEE Trans. Software Eng.*, vol. 46, no. 11, pp. 1176–1199, 2020.

[13] "Performance measurement | TensorFlow Lite." https://www.tensorflow.org/lite/performance/measurement.

[14] "Pytorch mobile.." https://pytorch.org/mobile/home/, 2019.

[15] "Tencent ncnn deep learning framework.." https://github.com/Tencent/ncnn, 2018.

[16] "Alibaba mnn deep learning framework.." https://github.com/alibaba/MNN, 2019.

[17] "Xiaomi mace deep learning framework.." https://github.com/XiaoMi/mace, 2017.

[18] "Snapdragon snpe deep learning framework.." https://developer.qualcomm.com/sites/default/files/docs/snpe/overview.html, 2017.

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

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

[21] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[22] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

[23] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[24] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-first AAAI conference on artificial intelligence*, 2017.

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

[26] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[27] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

[28] F. Iandola, M. Moskewicz, S. Karayev, R. Girshick, T. Darrell, and K. Keutzer, "Densenet: Implementing efficient convnet descriptor pyramids," *arXiv preprint arXiv:1404.1869*, 2014.

[29] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019.

[30] S. Singh and S. 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*, pp. 11237–11246, 2020.

[31] S. C. Yurtkulu, Y. H. Şahin, and G. Unal, "Semantic segmentation with extended deeplabv3 architecture," in *2019 27th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, IEEE, 2019.

[32] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*, pp. 21–37, Springer, 2016.

[33] Y. Yin, H. Li, and W. Fu, "Faster-yolo: An accurate and faster object detection method," *Digital Signal Processing*, vol. 102, p. 102756, 2020.

[34] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[35] L. Xu, X. Zhang, and Q. Dong, "Cluecorpus2020: A large-scale chinese corpus for pre-training language model," *arXiv preprint arXiv:2003.01355*, 2020.

[36] D. Kim, "A study of user data integrity during acquisition of android devices," 2013.

[37] M. Wang, S. Ding, T. Cao, Y. Liu, and F. Xu, "Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus," in *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pp. 215–228, 2021.

[38] D. Cai, Q. Wang, Y. Liu, Y. Liu, S. Wang, and M. 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*, pp. 31–36, 2021.

[39] Y. Guo, Y. Li, L. Wang, and T. Rosing, "Depthwise convolution is all you need for learning multiple visual domains," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 8368–8375, 2019.

[40] G. Sellers and J. Kessenich, *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional, 2016.

[41] F. Mues, "Optimization of opengl streaming in distributed embedded systems," 2020.

[42] G. Jo, W. J. Jeon, W. Jung, G. Taft, and J. Lee, "Opencl framework for arm processors with neon support," in *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, pp. 33–40, 2014.

[43] F. Jia, D. Zhang, T. Cao, S. Jiang, Y. Liu, J. Ren, and Y. Zhang, "Codl: efficient cpu-gpu co-execution for deep learning inference on mobile devices," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pp. 209–221, 2022.

[44] S. Chetoui and S. Reda, "Workload-and user-aware battery lifetime management for mobile socs," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1679–1684, IEEE, 2021.

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

[46] Y. Li, W. Wang, H. Bai, R. Gong, X. Dong, and F. Yu, "Efficient bitwidth search for practical mixed precision neural network," *arXiv preprint arXiv:2003.07577*, 2020.

[47] R. Wu, F. Zhang, Z. Zheng, X. Du, and X. Shen, "Exploring deep reuse in winograd cnn inference," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 483–484, 2021.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3301973

[48] R. Tornai and P. Fürjes-Benke, "Compute shader in image processing development," 2021.
[49] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *OSDI)*, pp. 265–283, 2016.
[50] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *arXiv preprint arXiv:1912.01703*, 2019.
[51] F. Plesinger, P. Nejedly, Z. Koscova, M. Rohr, I. Viscor, R. Smisek, A. Ivora, P. Leinveber, K. Curila, and C. Hoog Antink, "Deepplayer: An open-source signalplant plugin for deep learning inference," *Software: Practice and Experience*, vol. 53, no. 2, pp. 455–464, 2023.
[52] S. Boeschoten, C. Catal, B. Tekinerdogan, A. Lommen, and M. Blokland, "The automation of the development of classification models and improvement of model quality using feature engineering techniques," *Expert Systems with Applications*, vol. 213, p. 118912, 2023.
[53] S. Huang, N. Cai, P. P. Pacheco, S. Narrandes, Y. Wang, and W. Xu, "Applications of support vector machine (svm) learning in cancer genomics," *Cancer genomics & proteomics*, vol. 15, no. 1, pp. 41–51, 2018.
[54] T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho, K. Chen, *et al.*, "Xgboost: extreme gradient boosting," *R package version 0.4-2*, vol. 1, no. 4, pp. 1–4, 2015.
[55] F. Manessi, A. Rozza, and M. Manzo, "Dynamic graph convolutional networks," *Pattern Recognition*, vol. 97, p. 107000, 2020.
[56] H. Vandecasteele and G. Samaey, "Efficiency and parameter selection of a micro-macro markov chain monte carlo method for molecular dynamics," *arXiv preprint arXiv:2209.13056*, 2022.
[57] I. Leontiadis, S. Laskaridis, S. I. Venieris, and N. 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*, pp. 15–21, 2021.
[58] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. 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*, pp. 1–15, 2020.
[59] H. Yeo, C. J. Chong, Y. Jung, J. Ye, and D. Han, "Nemo: enabling neural-enhanced video streaming on commodity mobile devices," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pp. 1–14, 2020.
[60] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.
[61] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. 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*, pp. 2704–2713, 2018.
[62] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. 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*, pp. 389–400, 2018.
[63] Q. Zhang, X. Li, X. Che, X. Ma, A. Zhou, M. Xu, S. Wang, Y. Ma, and X. Liu, "A comprehensive benchmark of deep learning libraries on mobile devices," in *Proceedings of the ACM Web Conference 2022*, pp. 3298–3307, 2022.
[64] A. E. Abyane and H. Hemmati, "Robustness analysis of deep learning frameworks on mobile platforms," *arXiv preprint arXiv:2109.09869*, 2021.
[65] C. Luo, X. He, J. Zhan, L. Wang, W. Gao, and J. Dai, "Comparison and benchmarking of ai models and frameworks on mobile devices," *arXiv preprint arXiv:2005.05085*, 2020.
[66] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, *et al.*, "Mlperf training benchmark," *arXiv preprint arXiv:1910.01500*, 2019.
[67] X. Tang, S. Han, L. L. Zhang, T. Cao, and Y. Liu, "To bridge neural network design and real-world performance: A behaviour study for neural networks," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.
[68] H. Wang, H. Li, and Y. Guo, "Understanding the evolution of mobile app ecosystems: A longitudinal measurement study of google play," in *The World Wide Web Conference*, pp. 1988–1999, 2019.
[69] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, and L. Van Gool, "Ai benchmark: Running deep neural networks on android smartphones," in *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, pp. 0–0, 2018.
[70] P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann, "Automated algorithm selection: Survey and perspectives," *Evolutionary computation*, vol. 27, no. 1, pp. 3–45, 2019.
[71] S. Raschka, "Model evaluation, model selection, and algorithm selection in machine learning," *arXiv preprint arXiv:1811.12808*, 2018.
[72] B. Kutukcu, S. Baidya, A. Raghunathan, and S. Dey, "Contention-aware adaptive model selection for machine vision in embedded systems," in *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 1–4, IEEE, 2021.

Qiyang Zhang is a Ph.D. candidate in computer science at the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. He is also a visiting student in Distributed Systems Group at TU Wien from December 2022 to December 2023. His research interests include Mobile Edge Computing, Edge Intelligence.

Xiangying Che received a master's degree in Software engineering from the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. Her research interests include Cloud Computing and Mobile Edge Computing.

Yijie Chen received a bachelor's degree in Software engineering from Henan University in 2022. Currently, she is a postgraduate student in computer science at the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. Her research interests include Cloud Computing and Mobile Edge Computing.

Xiao Ma received her Ph.D. degree in Department of Computer Science and Technology from Tsinghua University, Beijing, China, in 2018. She is currently a lecturer at the State Key Laboratory of Networking and Switching Technology, BUPT. From October 2016 to April 2017, she visited the Department of Electrical and Computer Engineering, University of Waterloo, Canada. Her research interests include mobile cloud computing and mobile edge computing.

Mengwei Xu received the bachelor's and Ph.D. degrees from Peking University, Beijing, China. He is an Assistant Professor with the Computer Science Department, Beijing University of Posts and Telecommunications, Beijing. His research interests cover the broad areas of mobile computing, edge computing, and operating systems.

Schahram Dustdar is Full Professor of Computer Science heading the Research Division of Distributed Systems at the TU Wien, Austria. He is the co-editor-in-chief of the ACM Transactions on Internet of Things and the editor-in-chief of the Computing (Springer). He is also an associate editor of the IEEE Transactions on Services Computing, IEEE Transactions on Cloud Computing, ACM Transactions on the Web, and ACM Transactions on Internet Technology. He serves on the editorial board of the IEEE Internet Computing and IEEE Computer Magazine.

Xuanzhe Liu is an Associate Professor (with tenure) in the School of Computer Science at Peking University. His research interests mainly fall in service-based software engineering and systems. Most of his recent efforts have been published at prestigious conferences including WWW, ICSE, FSE, SIGCOMM, NSDI, MobiCom, MobiSys, SIGMETRICS, and IMC, and in journals including ACM TOSEM/TOIS/-TOIT/TWEB and IEEE TSE/TMC/TSC. He is a senior member of the IEEE and the ACM, and a distinguished member of the CCF. He serves as the corresponding author of this paper.

Shangguang Wang is a professor at the School of Computer Science, Beijing University of Posts and Telecommunications, China. His research interests include service computing, mobile edge computing, cloud computing, and satellite computing. He is currently serving as chair of IEEE Technical Community on Services Computing(TCSVC), and vice chair of IEEE Technical Community on Cloud Computing. He also served as general chairs or program chairs of 10+ IEEE conferences, advisor/associate editors of several journals such as Journal of Cloud Computing, Journal of Software: Practice and Experience, International Journal of Web and Grid Services, China Communications, and so on. He is a senior member of the IEEE, and Fellow of the IET.