Towards Ubiquitous Learning: A First Measurement of On-Device Training Performance

Dongqi Cai
State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, China
cdq@bupt.edu.cn

Qipeng Wang
Key Lab of High Confidence Software Technologies (Peking University), MoE, China
wangqipeng@stu.pku.edu.cn

Yuanqiang Liu
Key Lab of High Confidence Software Technologies (Peking University), MoE, China
yuanqiangliu@pku.edu.cn

Yunxin Liu
Institute for AI Industry Research (AIR), Tsinghua University, China
liuyunxin@air.tsinghua.edu.cn

Shangguang Wang
Shenzhen Research Institute, Beijing University of Posts and Telecommunications, China
sgwang@bupt.edu.cn

Mengwei Xu
State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, China
mwx@bupt.edu.cn

ABSTRACT

We are witnessing the emergence of *ubiquitous learning*, where each device (smartphones, wearables, IoTs, etc) can learn from their environments either alone or collaboratively. Such a new paradigm is enabled by deep learning techniques, or more specifically, on-device training. Given its popularity in the machine learning community, unfortunately, there are no systematic understandings of a critical question: how much cost does it take to train typical deep models on commodity end devices? Therefore, this work performs comprehensive measurements of on-device training with the state-of-the-art training library, 6 mobile phones, and 5 classical neural networks. Our measurements report metrics of training time, energy consumption, memory footprint, hardware utilization, and thermal dynamics, thus help reveal a complete landscape of the on-device training performance. The observations from the measurements help guide us to several promising future directions to efficiently enable ubiquitous learning.

CCS CONCEPTS

• **General and reference** → **Measurement**; • **Computing methodologies** → **Machine learning**; • **Human-centered computing** → **Ubiquitous computing**.

KEYWORDS

Measurement Study, Machine Learning, Ubiquitous Learning

ACM Reference Format:
Dongqi Cai, Qipeng Wang, Yuanqiang Liu, Yunxin Liu, Shangguang Wang, and Mengwei Xu. 2021. Towards Ubiquitous Learning: A First Measurement of On-Device Training Performance. In *5th International Workshop on Embedded and Mobile Deep Learning (EMDL'21), June 25, 2021, Virtual, WI, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3469116.3470009

EMDL'21 , June 25, 2021, Virtual, WI, USA

ACM ISBN 978-1-4503-8597-8/21/06…$15.00
https://doi.org/10.1145/3469116.3470009

1 INTRODUCTION

Deep learning technique is revolutionizing how edge devices interact with users or the world, including smartphones and IoT devices. Fueled by the increasingly powerful on-chip processors, the inference (or prediction) stage of deep learning is known to happen on edge devices without cloud offloading, making a case for low delay and data privacy protection [17, 18, 20–22]. Beyond inference, the training stage of deep learning is still commonly placed on data centers [8, 24] for its tremendous demand of massive training data and computing resources.

In recent years, however, we are witnessing the emergence of a new paradigm that directly leverages edge devices for model training, referred as "ubiquitous learning". Use cases of ubiquitous learning abound: input method [1], virtual assistant [2], item recommendation [13], etc. While the learning protocols may diversify, e.g., federated learning [12], split learning [15], and local transfer learning [19], they all rely on a fundamental system component: on-device training library. Given the highly constrained resources on edge devices and the huge resource demand of deep learning training stage, it's intuitive to ask *whether edge devices can really afford training modern NN models and if so, do current libraries efficiently support that?*

While ubiquitous learning is attracting increasing attentions from research communities, most existing efforts focus on the algorithm level [11]. They address the challenges like non-iid data distribution, huge data transmission among cloud and devices, and data privacy protection. Their proposed approaches are mostly evaluated via simulation [23]. A few recent studies use Raspberry Pi (RPI) devices to study ubiquitous learning [7], but often in an incomprehensive way and omit smartphones, the killer use case for ubiquitous learning. In a nutshell, **there's a lack of system-level understanding of how ubiquitous learning is supported**

Testing platform	Training library	Training time (ms)		
		BS = 1	BS=2	BS=4
Samsung Note 10	MNN	516	812	1365
	DL4J	3,032	6,129	OOM
RPI 3B+	MNN	6698	10,651	OOM
	TensorFlow	10,468	14,157	27,574
	PyTorch	48,274	79,097	OOM

Table 1: A brief comparison of training libraries on edge devices. Time measured as training one batch of MobileNet-v2. "BS": batchsize. All experiments run on CPU.

Device	Specifications	Yr.
Redmi Note 9 Pro	Snapdragon 720G, 6GB RAM	2020
Xiaomi MI 9	Snapdragon 855, 6GB RAM	2019
Huawei Mate 30	Kirin 990, 8GB RAM	2019
Meizu 16T	Snapdragon 855, 6GB RAM	2019
Samsung S8+	Snapdragon 835, 6GB RAM	2017
Huawei Honor 8	Kirin 950, 3GB RAM	2016

Table 2: Testing devices.

by commodity edge devices and state-of-the-art training libraries.

For the first time, we present a comprehensive measurement study of on-device training performance. We build a full-fledged benchmark suite based on MNN [10], the state-of-the-art training library for edge devices. The benchmark includes 5 classical NN models, and can report all-sided training performance metrics including training latency, energy consumption, memory footprint, hardware utilization, and thermal dynamics. We then carry out experiments on 6 different Android devices. We also investigate the impacts of system parameters (CPU cores and frequency) on the training performance.

Our experimental results shed light on future optimizations to efficiently enable ubiquitous learning. For example, we find that MNN is still at a preliminary stage and its implementation is far from being efficient as compared to its inference functionality that has been optimized for many years. The memory constraint imposed by both the physical RAM and OS's resource allocation strategies severely limits the batchsize of training and therefore potentially hurt the convergence performance. In addition, tuning the system parameters opens rich tradeoff among multiple on-device training metrics. The training libraries should adapt those parameters to cater for each concrete scenario.

Open source The full benchmark suite and measurement results used in this work are available here[1].

2 MEASUREMENTS

2.1 Experiment setups

On-device training library We first investigate the ML libraries that support training on typical edge platforms. (1) On Android phones[2], only MNN [10] and DL4J [3] have off-the-shelf support to train NNs locally. (2) On RPI devices, MNN and many other traditional ML libraries (e.g., TensorFlow and PyTorch) enable local training. To study the status quo of on-device training, we first briefly compare the performance of those frameworks. As shown in Table 1, on both Android phone and RPI, MNN achieves better performance than its competitors. It is not surprising as MNN is lightweight, highly-optimized for edge devices, and already widely adopted in popular productions of Alibaba Inc. Therefore, the rest of this study focuses MNN as the major testing objective.

Metrics The processing time (for both inference and training) is directly logged in our benchmark suit. The temperature information is obtained through `dumpsys` tool. Other system information, including hardware utilization CPU, energy consumption, and memory footprint are obtained through Android's virtual filesystem (e.g., `/sys` and `/proc`).

Devices We carry out our experiments on 6 devices with diverse SoC models and computing capacity, as summarized in Table 2. By default, all experiments run on mobile CPU with 4 cores. We observe that the training tasks will always be scheduled to the big cores that operate at highest frequency as determined the OS.

Models We test 5 classical CNN models in our experiments: LeNet (2 convs, 3.2K parameters), AlexNet (5 convs, 61M parameters), MobileNetv2 (53 convs, 3.4M parameters), SqueezeNet (18 convs, 411.2K parameters), and GoogLeNet (22 convs, 6.8M parameters).

Datasets We use two different datasets for the experiments: MNIST dataset (70000 images, 10 classes, $28 \times 28 \times 1$ inputsize) and a subset of ImageNet (3200 images, 4 classes, $224 \times 224 \times 3$ inputSize).

2.2 Training time and breakdown

Overall latency Figure 1 illustrates the inference and training time of each model on Meizu 16T. Our primary observation is that the training time is much larger than the inference time (up to 17.8× gap with MobileNet and batchsize 16). Such a measured gap is significantly larger than the theoretical FLOPs gap (around 3×). This is because, MNN was first released as an inference-only engine and thus highly optimized for that. The training support was first added in late 2019, and the implementation of many operators is to be improved. Indeed, due to its huge number of operators and dynamic parameter update, training functionality is much more difficult to be efficiently implemented than inference functionality. For example, many sophisticated optimizations designed for inference operators like winograd convolution cannot be applied to deconvolution in backward pass. In addition, the one-shot pre-processing of weight tensors for inference speedup cannot be used in training either as the weights will be updated during backward propagation.

We also observe that with larger batchsize, the training time increases more significantly than the inference time. This is mainly because, as confirmed by Figure 2, the CPU usage during training is already close to maximal with relatively small batchsize (400% as 4 CPU cores are used by default), while during inference using

[1] https://github.com/UbiquitousLearning/Benchmark-On-Device-Training

[2] There are other libraries like CoreML supporting NN training on iOS devices, but they are omitted in this study since iOS devices are difficult to be instrumented to obtain low-level measurement results.

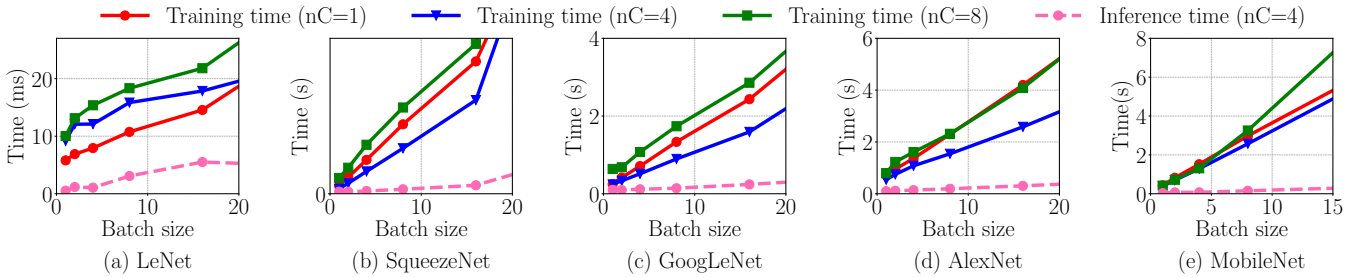


Figure 1: The training and inference time with various batchsize on Meizu 16T. "nC": number of CPU cores used for training/inference. The training task is scheduled to big cores first by the OS.

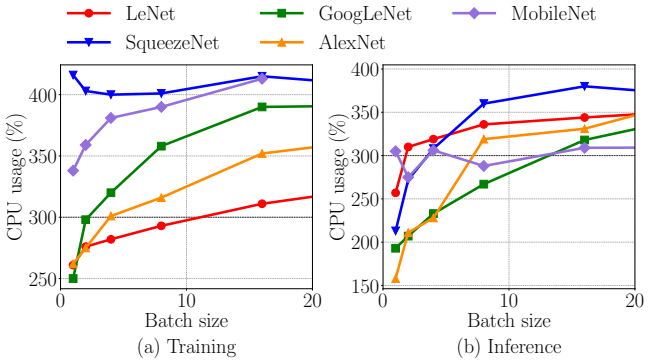


Figure 2: CPU usage of training/inference on MI 9.

larger batchsize can noticeably increase the CPU usage thus the inference latency increases sublinearly.

CPU core number MNN allows developers to configure the number of CPU cores to be used. Figure 1 also illustrates the training time with different core numbers used (1, 4, and 8). Our main observation is that multiple cores can often speed up the training as compared to one single core. The only exception is on LeNet with small batchsize (≤16) where using only one core exhibits the best performance. This is because LeNet is a tiny model that can hardly benefit from multi-core parallelism. On the other hand, more CPU cores don't always bring improvements. In fact, distributing the training into all 8 CPU cores often leads to the highest training time. This is mainly because mobile CPU cores are asymmetric, i.e., big.LITTLE architecture, and MNN's workloads partitioning strategies is imperfect [16]. The observation is consistent with inference stage, where we observe that 4 CPU cores achieve the lowest latency in most cases.

Cross-device comparison Figure 3 further compares the training time on different devices. Overall, the performance disparity across devices is pervasive and can be up to 4.0× (MobilenetV2 model among Meizu 16T and Samsung S8+). For collaborative learning across mobile devices like federated learning, such heterogeneous computing capacity needs to be considered to better synchronize the training pace. Interestingly, we also observe that different devices have distinct "model affinity". For example, SqueezeNet trains fastest on Meizu 16T while AlexNet trains fastest on Huawei Mate 30.

Latency breakdown We further break down the end-to-end training time into the lowest operator level. The results are summarized in Figure 4. `Raster` and `MatMul` are the two most time-consuming operators. In MNN, `Raster` operator is a unified implementation of all traditional operators that are related to tensor shape transformations, including `slice`, `concat`, `reshape`, `broadcast`, etc. For example, the convolution operation in MNN is implemented with 3 `Raster` (two to convert input and filters, one to convert output) and 1 `MatMul`. After communicating with the developers of MNN, we find that the reason MNN doesn't implement those operators individually is mainly because using one operator (`Raster`) to represent those similar operations can help them focus on optimizing that particular operator and thus greatly reduce the developers' efforts. However, such a simplified design inevitably sacrifices the running performance of operators.

2.3 Memory footprint

NN training is known to be memory hungry. Figure 5 shows the memory footprint of each tested model with diverse batchsizes on Mate 30. As observed, with a typical batchsize 16, the memory usage of the tested models (except LeNet) is about a few GBs. For those models, 16–64 is the maximal batchsize that a high-end mobile phone (e.g., MI 9 with 6GB RAM) can support. This is much more than the memory requirement of inference stage (e.g., a few hundred MBs for MobileNet) because, during inference, the input is usually fed into model one by one and the intermediate tensors don't need to be preserved for backward propagation.

Noting that our experiments are carried out in a "clean" environment without other co-running applications. In practice, however, requesting such a high amount of memory may cause other applications to be swapped out, or raises the risk of the training app itself to be moved out of memory by the OS. Therefore the memory usage of on-device training task shall be more strictly confined, indicating that an even smaller batchsize can be eventually employed, e.g., 4 for MobileNet to keep the memory usage under 1GB. Such a small batchsize has been demonstrated to be not enough to guarantee the model convergence [14]. For example, our micro experiments show that using batchsize smaller than 128 will cause significant accuracy drop of MobileNet-v2 and SqueezeNet on CIFAR-100 datasets.

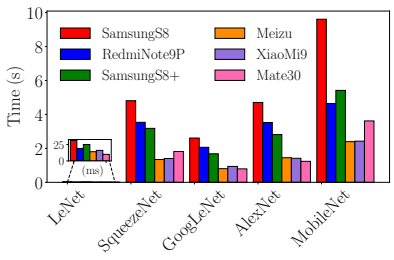


Figure 3: Training time on different mobile devices (batchsize=8).

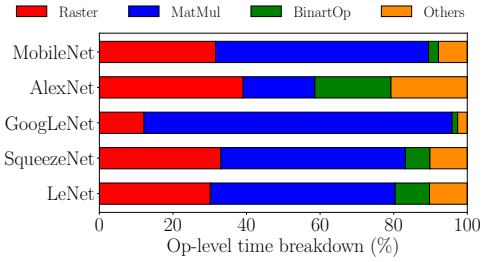


Figure 4: Op-level breakdown of training time on Xiaomi MI 9.

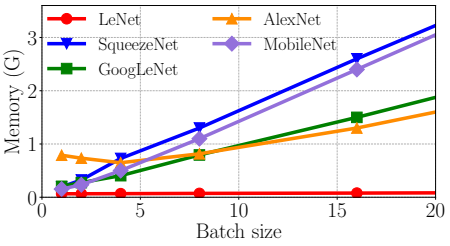


Figure 5: Peak memory usage during training.

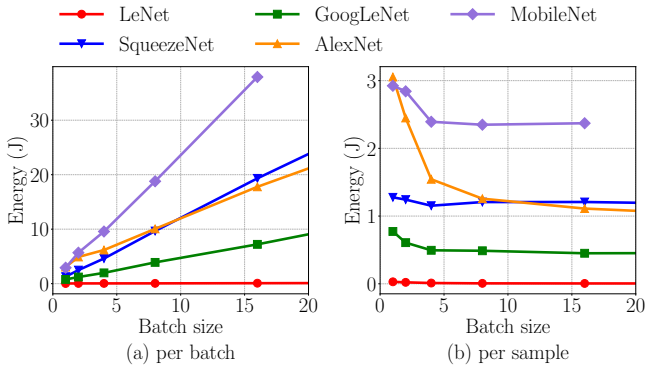


Figure 6: Per-batch and per-sample energy consumption of training on Redmi Note 9 Pro.

CPU Conf.	Time (s)			Energy (J)		
	H	M	L	H	M	L
1×	4.2	5.4	10.8	10.6	8.0	6.9
Big 2×	2.6	3.2	6.4	8.9	7.7	7.0
4×	2.0	3.3	8.4	7.1	8.7	8.2
1×	25.0	33.9	57.8	10.4	7.2	3.1
Small 2×	13.3	18.0	31.8	10.1	8.4	4.8
4×	8.0	11.0	52.3	11.4	9.6	8.2
Hybrid 8×	3.8	6.5	50.4	13.4	13.9	14.4

Table 3: Training performance with different CPU configurations on Meizu 16T. "H": highest frequency (2.4GHz/1.8GHz for big/little core); "M": medium frequency (1.6GHz/1.2GHz for big/small core); "L": lowest frequency (0.7GHz/0.6GHz for big/little core). Model: Alexnet. Batchsize: 16. Numbers in red/gray indicate the best/worst performance.

2.4 Energy consumption

Figure 6 summarizes the energy consumption per batch (a) and per sample (b) with different models and batchsizes. In this experiment, we run each model for 20 batches and subtract the energy consumption during that running period by the baseline energy when no training task is running. The increased output power during training is multiplied by the duration of training to calculate the energy consumption.

As observed, the per-batch energy consumption significantly increases with larger batchsize mainly because of the increased training time. However, the per-sample energy consumption decreases with larger batchsize (e.g., from 1 to 16 for GoogLeNet). This is because the training time doesn't linearly scale with the batchsize since larger batchsize benefits the intra-operator parallelism (§2.2). It indicates that to iterate over a local dataset, using larger batchsize can be more energy-efficient.

2.5 Impacts of CPU parameters

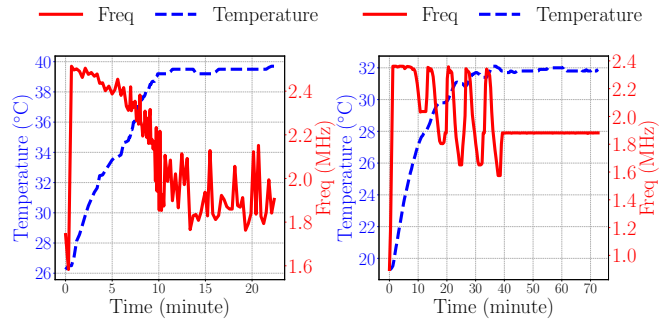
We then study how system configurations (CPU in our case) affect the training performance. We vary the CPU core numbers (1×/2×/4× big cores, 1×/2×/4× small cores, or 8 hybrid cores) and the frequency for each core (highest, medium, lowest) on Meizu 16T with Snapdragon 855 (4 big cores + 4 small cores). The results are summarized in Table 3.

For training time, using 4 big cores with the highest frequency achieves the best performance. The result is consistent with Figure 1 and the reasons are explained in Section 2.2. In consideration of energy consumption, however, using only one small core with the lowest CPU frequency leads to the lowest usage. Its energy consumption is only 43.7% of the optimal case of training time, despite it runs 28.9× slower.

The observation indicates a large space of tradeoff between training time and energy consumption. In reality, both metrics play key roles in ubiquitous learning. Taking federated learning as an example, developers expect the on-device training to be completed shortly therefore the global model converges fast. Meanwhile, the energy consumption shall be minimized to not compromise user experience. Considering the hardware heterogeneity across devices and dynamics of network bandwidth, it's nontrivial to choose an appropriate system configuration for each device.

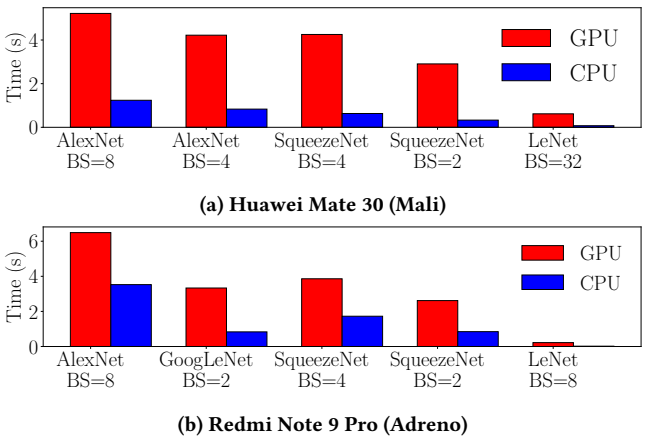
2.6 Thermal dynamics

To reach a usable accuracy, the training phase often takes a substantial period of time, e.g., minutes for each round of federated learning [4] or even hours for continuous local transfer learning [19]. Such a long duration of intensive computation may lead to thermal issues and therefore the CPU frequency change due to dynamic voltage and frequency scaling (DVFS) even without any other applications running. Thus we investigate into the thermal dynamics



(a) Meizu 16T (b) Samsung S8+

Figure 7: The temperature and CPU frequency dynamics during on-device training.



(a) Huawei Mate 30 (Mali)

(b) Redmi Note 9 Pro (Adreno)

Figure 8: Comparing the training time of two mainstream GPU models to CPUs. "BS": batchsize.

and their impacts on the CPU frequency (averaged across all 4 used cores) on two devices and illustrate the results in Figure 7.

On both tested devices, we observe the temperature rising and the CPU underclocking. Though, the specific behavior differs on different devices. For instance, on Meizu 16T the device temperature rises more sharply from 25°C to 39°C in 10 minutes, and the average CPU frequency gradually drops from 2.5GHz to 1.8GHz. On Samsung S8+, it takes around 30 minutes of training to raise the temperature from 20°C to 32°C, while the CPU frequency jitters for a while and stabilizes around 1.8GHz. Such heterogeneous thermal dynamics may complicate the ubiquitous learning process, e.g., the device selection in federated learning.

2.7 Mobile CPU vs. Mobile GPU

We also study the training performance of MNN on mobile GPUs. Since MNN's GPU support for training is not complete, we manually add support for several missing operators like `onehot`. As summarized in Figure 8, both Mali (Huawei Mate 30) and Adreno (Redmi Note 9P) run much slower than CPU (1.8×–12.0×).

According to our experience in tuning operators for on-GPU training, one of the primary unique challenges is batchsize, a new variable introduced in training that affects the tensor shape and thus a lot of other aspects at implementation level such as memory layout and alignment. For inference engines like tflite, however, the most common use case is batchsize as 1, and a larger batchsize can be simply regarded as a sequence of single samples aggregated. Therefore, almost every inference engine is specifically tuned for that case, markedly simplfying the development of GPU kernels.

Another reason is that, as discussed in Section 2.2, MNN splits common operators into smaller, more basic operators. Many of those basic operators like `Raster` are memory-intensive, so cannot benefit from high parallelism of GPUs.

3 CONCLUSIONS AND FUTURE WORK

In this work, we perform first-of-its-kind measurements of on-device training performance on commodity smartphones. The results lead us to several interesting findings, emphasizing that we are at the dawn of "learning everywhere and anytime". It calls for more research efforts, in both theory and system aspects, to enable such ubiquitous learning paradigm. Here we highlight some of the promising future directions.

• **Generating efficient operators** While being the state-of-the-art training library for edge devices, our experiments show that MNN's performance is still far from optima (Section 2.2 and Section 2.7). The reason is that tunning the operators is both difficult and time-consuming even for ML experts due to the variable batchsize and the asymmetric multiprocessing feature on mobile SoCs. Since the early release of TensorFlow Lite in 2017, we have witnessed the emergence and continuously improved performance of (new) inference engines for mobile CPU, GPU, DSP, and even NPUs. As comparison, on-device training libraries like MNN are still at very preliminary stage from the perspective of the open-sourced software ecosystem. One potential solution to speed up or even skip the labor-intensive tuning of NN operators is to leverage automatic tensor compiler [5] to generate efficient operators for training.

• **Memory optimizations** The results in Section 2.3 show that, constrained by the physical RAM and the memory allocation strategies of OS, current mobile devices only allow a small batchsize in training typical CNN models. Such a limitation may harm the convergence performance (both accuracy and wall clock time). Therefore, memory optimizations are needed to enable training on large-enough batchsize. We may retrofit existing techniques invented for datacenter GPUs memory, e.g., memory swapping [9] and checkpointing [6]. Those techniques, however, cannot be directly applied as mobile SoCs have different architecture than datacenter-scale GPUs (e.g., integrated memory for heterogeneous processors), and they may impose high computation overhead mobile devices cannot afford.

• **Tuning system parameters** As shown in Section 2.5, system parameters like CPU cores and frequency opens rich tradeoffs for on-device training performance, i.e., training time and energy consumption. Both of these metrics play key role in designing an efficient ubiquitous learning protocol. For example, in federated learning, many devices equipped with heterogeneous processors may leverage different CPU frequencies to keep a synchronized pace

while reduce the overall energy consumption. Choosing the optimal configuration is challenging, as our experiments show that it depends on model strucutre, batchsize, hardware specification and status, etc.

ACKNOWLEDGMENTS

This work was supported by National Key R&D Program of China under grant number 2020YFB1805500, the Fundamental Research Funds for the Central Universities, and National Natural Science Foundation of China under grant numbers 62032003, 61922017, and 61921003.

REFERENCES

[1] Federated learning: Collaborative machine learning without centralized training data. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html, 2017.

[2] How apple personalizes siri without hoovering up your data. https://www.technologyreview.com/2019/12/11/131629/apple-ai-personalizes-siri-federated-learning/, 2019.

[3] Deep learning for java. https://deeplearning4j.org/, 2021.

[4] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, H Brendan McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, pages 578–594, 2018.

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[7] Yansong Gao, Minki Kim, Sharif Abuadbba, Yeonjae Kim, Chandra Thapa, Kyuyeon Kim, Seyit A Camtepe, Hyoungshick Kim, and Surya Nepal. End-to-end evaluation of federated learning and split learning for internet of things. *arXiv preprint arXiv:2003.13376*, 2020.

[8] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 485–500, 2019.

[9] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *ASPLOS*, pages 1341–1355, 2020.

[10] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, et al. Mnn: A universal and efficient inference engine. *arXiv preprint arXiv:2002.12418*, 2020.

[11] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*, 2019.

[12] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[13] Chaoyue Niu, Fan Wu, Shaojie Tang, Lifeng Hua, Rongfei Jia, Chengfei Lv, Zhihua Wu, and Guihai Chen. Billion-scale federated learning on mobile clients: A submodel design with tunable privacy. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[14] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[15] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. Split learning for health: Distributed deep learning without sharing raw patient data. *arXiv preprint arXiv:1812.00564*, 2018.

[16] 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.

[17] 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.

[18] Mengwei Xu, Yunxin Liu, and Xuanzhe Liu. A case for camera-as-a-service. *IEEE Pervasive Computing*, 2021.

[19] 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.

[20] Mengwei Xu, Feng Qian, Mengze Zhu, Feifan Huang, Saumay Pushp, and Xuanzhe Liu. Deepwear: Adaptive local offloading for on-wearable deep learning. *IEEE Transactions on Mobile Computing*, 19(2):314–330, 2019.

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

[22] 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.

[23] Chengxu Yang, QiPeng Wang, Mengwei Xu, Shangguang Wang, Kaigui Bian, and Xuanzhe Liu. Heterogeneity-aware federated learning. *arXiv preprint arXiv:2006.06983*, 2020.

[24] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis CM Lau, Yuqi Wang, Yifan Xiong, et al. Hived: Sharing a {GPU} cluster for deep learning with guarantees. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 515–532, 2020.