

Article

Overlay-ML: Unioning Memory and Storage Space for On-Device AI on Mobile Devices

Cheolhyeon Kwon ¹ and Donghyun Kang ^{2,*} ¹ Department of Computer Engineering, Changwon National University,
Changwon-si 51140, Republic of Korea; cheolhyeon@gs.cwnu.ac.kr² Department of Computer Engineering, College of IT Convergence, Gachon University,
Seongnam-si 13120, Republic of Korea

* Correspondence: donghyun@gachon.ac.kr

Abstract: Recently, the technologies of on-device AI have been accelerated with the development of new hardware and software platforms. Therefore, many researchers and engineers focus on how to enable ML technologies on mobile devices with limited hardware resources. In this paper, we revisit on-device ML designed to support ML technologies on mobile devices and describe the three challenges when using on-device ML in detail. Then, we propose a new data management policy, called *Overlay-ML*, which efficiently solves two challenges that we discovered. Especially, we designed *Overlay-ML* to work in the application space with two key ideas. The first key idea is to extend the limited memory space using the usable space of the underlying storage device. The second key idea is to provide data transparency, which hides where the data is stored so that running ML models think the data is stored in the same place. For evaluation, we implemented an image detection application based on TensorFlow Lite which is a well-known on-device ML framework, and modified it to enable the features of *Overlay-ML*. All evaluation was performed on two state-of-the-art smartphones that are high-end embedded devices. Our evaluation results clearly show *Overlay-ML* can effectively prevent unexpected termination by Android OS and present a good loss value in real-world workload.

Keywords: deep learning; image classification; data and memory management; embedded devices; embedded operating system

**Citation:** Kwon, C.; Kang, D.Overlay-ML: Unioning Memory and Storage Space for On-Device AI on Mobile Devices. *Appl. Sci.* **2024**, *14*, 3022. <https://doi.org/10.3390/app14073022>Academic Editor: Francesco
Liberato Cappiello

Received: 12 March 2024

Revised: 29 March 2024

Accepted: 2 April 2024

Published: 3 April 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Nowadays, many researchers and engineers have been leveraging various machine learning (ML) techniques to improve the performance of image processing and pattern recognition [1–4]. Especially, companies developing image sensors have started to introduce their new products to ensure the best image detection and performance because image sensors are considered the eyes of IoT devices [5]. In addition, the trends of image-sensing technologies and the corresponding ML techniques have shifted from low-end devices that just collect raw data from sensor devices to high-end and smart devices, which include more computation power and functions. For example, some studies focused on deep learning (DL) technologies that enable automated image classification in vehicles [4]. In fact, these are possible because the computing power of embedded devices (e.g., RISC-V [6], Raspberry Pi [7], smartphones, etc.) is growing to the point where it is sufficient to run ML techniques. Some researchers focused on optimizing on-device ML technologies [8–10]. Unfortunately, they never focused on the limitations of mobile devices while running ML models and did not utilize real-world embedded devices (e.g., smartphones or tablets) for their experiments.

Meanwhile, the emergence of on-device ML [11–13] blurs the boundary ML technique on embedded devices; on-device ML executes training and inference procedure with the software help of ML instructions. For example, Google recently announced a new powerful package, called *ML Kit*, for mobile developers using Google machine learning [14]. It provides easy-to-use APIs that are supposed to detect barcodes, text, faces, and objects in

video and image files on mobile devices. TensorFlow Lite was designed to allow enabling on-device ML using transfer learning (TL), which reuses knowledge of a pre-trained model for intelligent services on embedded devices; the pre-trained model deals with full-scale datasets [12]. Thanks to these useful efforts and advances, we believe that it will be able to run more and more ML in mobile environments, such as data sensing on IoT platforms.

However, there are some challenges in using ML on embedded devices. For example, the capacity issue of memory on embedded devices still remains because of limited space where memory chips are placed; an application running ML technologies on embedded devices still suffers from memory pressure. Therefore, an application developed with TensorFlow Lite [12] is unintentionally killed when it requires more memory space than the available memory space per application. The reason behind this is Android terminates an application that uses memory space over the pre-defined threshold (i.e., 256 MB) because of the fairness of memory resources. Unfortunately, such a killing issue can appear and is unavoidable for general applications on any embedded devices running the Android operating system (OS). A promising alternative to solve this problem guided by Google is turning on “*largeHeap*” option that extends the pre-defined threshold to 512 MB. However, the expanded memory footprint is too small to adequately run on-device ML because such an application commonly requires gigabyte volumes of memory.

In this paper, we propose Overlay-ML which includes a new management policy to run on-device ML for image processing and pattern recognition even with memory pressure and limited memory space. To the best of our knowledge, it is the first solution to handle the memory issue for supporting applications with on-device ML. The key design concept of Overlay-ML is that it simultaneously handles tensors both on memory and storage media on embedded devices using *application-level swapping* and performs *multi-layer shuffling* to avoid overfitting issues of learning workloads. In this paper, we make the following contributions:

- In order to understand the memory pressure issue, we briefly introduce the internal procedure of TensorFlow Lite [15] which is widely used for on-device ML. Then, we describe that an application implemented with TensorFlow Lite can lead to undesirable termination by Android.
- We propose a design of Overlay-ML that helps to stably run applications developed by on-device ML without undesirable termination and show how the Overlay-ML works in detail.
- We implement Overlay-ML at the application level and compare it with the default application of TensorFlow Lite on top of the two latest smartphones, including Pixel 6 Pro and Galaxy S20+.

The rest of this paper is organized as follows: Section 2 introduces background and related work to understand our work. Section 3 explains how Overlay-ML works on embedded devices in detail. Section 4 shows our evaluation results on high-end smartphones. Finally, Section 5 concludes this paper.

2. Background and Related Work

In general, high-end embedded devices are equipped with a large amount of memory, but it is still several orders of magnitude smaller than servers or desktops. Therefore, the traditional ML model that runs on machines with rich hardware supports (e.g., CPUs, GPUs, and memory) poses unexpected challenges on embedded devices. In this section, we first introduce the trends of the sensor industry to understand the position of our work and its effectiveness in sensor environments. We briefly describe training and validation procedures in traditional ML and introduce three challenges in designing on-device ML. We also discuss prior efforts to optimize inference services on embedded devices in detail.

2.1. Traditional ML Model

The traditional ML model is composed of training and validation procedures. Such training and validation procedures are well-known as resource-expensive tasks in that they try to fully utilize hardware resources as much as possible. In this section, we first briefly

introduce how the ML model works and describe how hardware resources are utilized from the ML model in detail. In general, the training and validation procedures can be organized with the following four steps (see Figure 1) [16]:

- **Data-load:** In this step, a dataset for training and validation is loaded from network or local storage devices.
- **Pre-processing step:** The raw data in the loaded dataset is individually transformed into a tensor; then, each tensor is randomly shuffled to avoid the *overfitting problem*.
- **Training step:** It finds and learns correct answers using an ML model where weight values for inference are updated via forward and backward propagation.
- **Validation step:** It infers an acceptable prediction based on the weight values passed from the prior step.

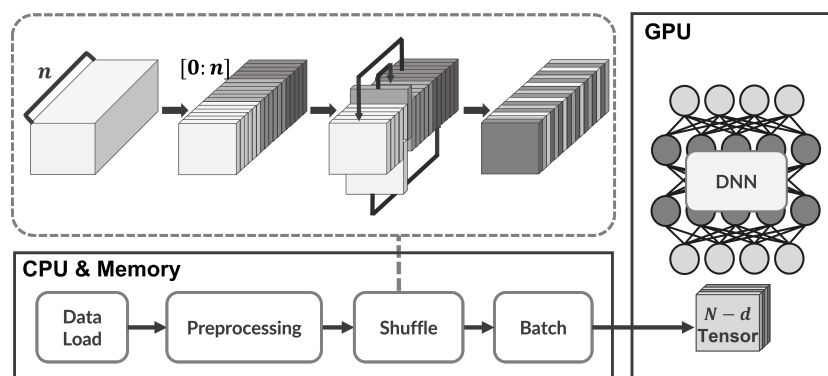


Figure 1. The sample of the line detection.

Note that the first two steps are mainly executed on CPUs and the remaining two steps are recommended to run on GPUs so as to speed up the time-consuming calculation. In addition, all steps require a large amount of memory space to handle their own computation. Therefore, reducing the rich hardware utilization is the first challenge in porting a popular ML procedure to embedded devices; we call this **(C1) challenge of hardware limitation**. In other words, all steps for an intelligent service on embedded devices must be performed using only arm-based CPUs and a limited capacity of memory.

2.2. On-Device ML Functionalities

To address the aforementioned challenge, some developers and researchers have focused on ML frameworks and modified them to be lightweight and efficient running on embedded devices. For example, TensorFlow [17,18] and PyTorch [19] frameworks have been ported to mobile frameworks (e.g., TensorFlow Lite [12,15] and PyTorch Mobile [13]) to meet the increasing need for on-device ML. In other words, they successfully support running most ML models on embedded devices (e.g., RISC-V [6], Raspberry Pi [7], smartphones, etc.). Unfortunately, the current frameworks for on-device ML cannot cover all kinds of ML functionalities supporting servers or desktops yet; we call it **(C2) challenge of functionality limitation**. Of course, the impact caused by this challenge may be insignificant because it is unnecessary to run all ML models on embedded devices equipped with poor computation power. However, recent ML trends have been moved toward embedded ML in that sensing data can be transformed near the physical location of either the user or the source of the data.

Meanwhile, most applications for on-device ML adopt a hybrid fashion that performs the training and validation steps with the transfer learning (TL) technique [20,21]. TL is one of the well-known ML techniques because it can decrease the peak usage of hardware resources (e.g., CPUs, GPUs, and memory) and save ML's time by sharing the knowledge of a pre-trained model based on a large dataset. However, an application based on the TL technique can be terminated when the memory requirements are over the limited memory threshold of the operating systems; we describe the reason in the next section.

2.3. Memory Management Policy of Embedded Operating System

Today, most embedded devices commonly adopt Linux family as their operating system [22]. For example, high-end embedded devices that have sufficient DRAM memory and CPU power commonly use Android OS, while low-end devices with limited hardware resources employ customized Linux, such as Raspberry Pi OS [7]. We believe that the use of Linux will be growing over time. This is because Linux can be easy to customize codes (i.e., Kernel) according to requirements and expand to support a diversity of hardware resources, such as ZigBee [23], IoT Hub [24], etc.

In this paper, we focus on Android OS and its memory management policy; it is widely used and follows the basic functionalities of Linux. Android OS guarantees the available memory space for each application by allocating memory space in an incremental way to store application data. It manages free memory space by killing an application if the accumulated memory usage exceeds the available memory limitation of one application (i.e., 256 MB). But, it is a very interesting termination because it occurs even if available memory space is enough. In other words, this termination is not triggered by *low memory killer daemon* (LMKD) that secures memory space under high memory pressure and low available memory space [25,26].

Meanwhile, Android gives us *largeHeap* option as a guide to solve this problem. This option works because of two reasons. First, the *largeHeap* option extends the total memory usage of an application to 512 MB [27]. Second, most data generated from on-device ML is saved on the *heap* area of a process. Unfortunately, some applications using on-device ML can require more memory space than 512 MB of memory space. In this case, the applications will be terminated when exceeding the total memory usage over 512 MB regardless of the *largeHeap* option. To confirm the effect of the option and the reason why an application is killed, we performed an experimental evaluation. For evaluation, we first implemented a simple application where an image is recognized and classified using on-device ML [15]. Figure 2 shows our evaluation results and memory usage that was monitored by *Symple System Monitor application*. As shown in Figure 2 (right), when the total memory usage of the application exceeds 512 MB, the application is killed with the following message “Out-Of-Memory Error” even though available memory space is enough. To understand the reason, we studied and found out that the error message means the application cannot allocate additional memory space because it had consumed its available memory space; we call it **(C3) challenge of software limitation**.

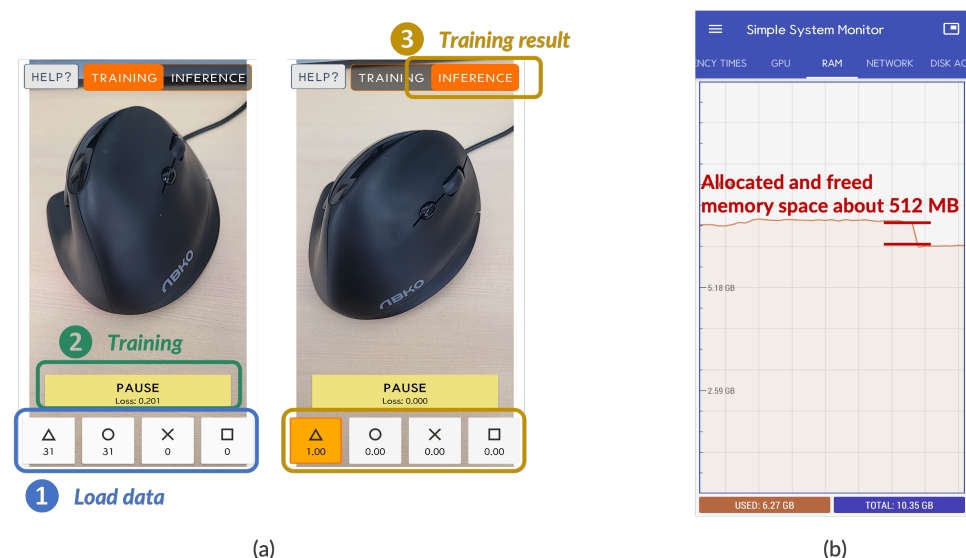


Figure 2. The results of experiment evaluation. An application developed by TensorFlow Lite (a) and its memory usage collected by a system monitoring application (b).

In summary, on-device ML still suffers from three challenges. We believe **C2** will naturally resolve over time because many researchers and engineers try to stably transform all functionalities of ML running on server or desktop environments. Therefore, we are motivated by a simple question: Can we address the **C1** and **C3** challenges with small trade-offs?

3. The Design of Overlay-ML

To efficiently run on-device ML on all kinds of embedded devices, we design a new data and memory management scheme, called Overlay-ML. The key idea of Overlay-ML is to extend short of memory by utilizing the space of underlying storage devices; the extension is provided with transparency that is invisible to the running ML frameworks. Note that Overlay-ML categorizes the procedures of on-device ML into two parts: overlay-write and overlay-shuffle-read, and includes a total of seven operations to correctly complete training and validation procedures.

3.1. Overlay-Write

Figure 3 shows the overall architecture and flows of Overlay-ML on embedded devices when on-device ML runs for image processing and pattern recognition; Overlay-ML is designed to work in the application space. As mentioned before, Android OS does not allow for an application to utilize memory space over the pre-defined threshold (i.e., 256 MB or 512 MB). To avoid unexpected termination of an application by Android OS, Overlay-ML provides two write paths: a *fast write path* and a *slow write path*. Also, Overlay-ML monitors how much memory space an application running on-device ML is using to determine which write path runs. In the data-load step, the fast write path follows the traditional store procedure of on-device ML, thus it is always triggered first. If the memory usage of the monitored application is lower than the threshold of Android OS, Overlay-ML keeps using the fast write path to store data in the main memory space (denoted by ❶). If the memory usage of the application exceeds the threshold, Overlay-ML starts to store data on the underlying storage device via the slow write path (denoted by ❷). Now, data generated by on-device ML can be stored whether in the main memory or the underlying storage device space. Therefore, Overlay-ML tracks and manages the location where data is stored by using a mapping table. In other words, data generated by on-device ML before the memory threshold is stored in the main memory space, and data beyond the threshold is stored in the underlying storage device space.

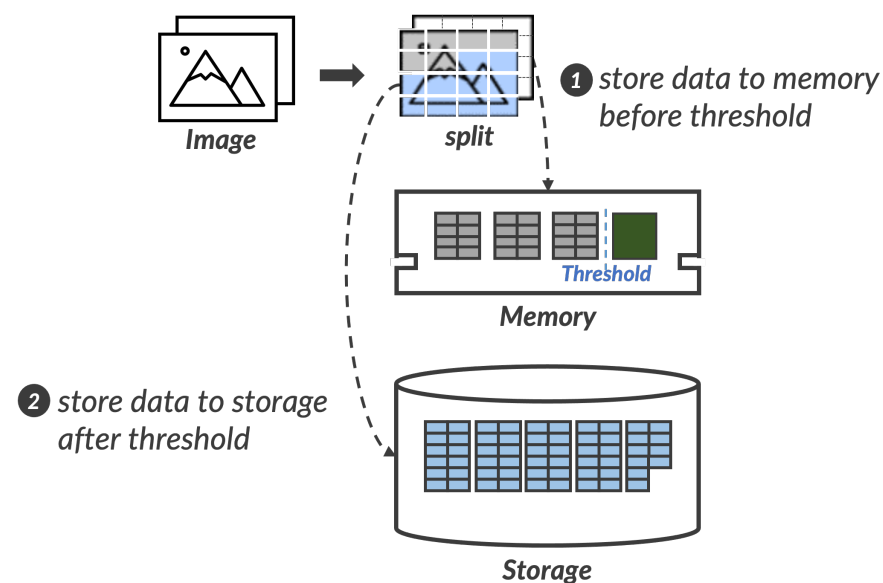


Figure 3. The overall flows of the overlay-write operation in Overlay-ML.

On the other hand, since two different applications using on-device ML can run concurrently, Overlay-ML should isolate data generated by one application from others. To support this isolation, Overlay-ML stores data in the form of a dedicated file that is generated per application.

3.2. Overlay-Shuffle-Read

Now, we describe how Overlay-ML handles data that is partially stored whether in main memory or storage device space to correctly perform inference on embedded devices. Figure 4 shows the overall flows of overlay-shuffle-read operation in Overlay-ML. After the data-load step, the pre-processing step is first triggered to make tensors and randomly shuffle them (denoted by ③). Unlike traditional on-device ML, Overlay-ML should scan data both in the main memory and the underlying storage device space. At this step, Overlay-ML randomly selects data regardless of where it is stored to avoid overfitting or bias issues. We call this *hybrid shuffling* in the rest of this paper. In other words, shuffling only data belonging to the main memory space can increase the loss value, which represents the sum of errors in the ML model, because there is not enough sample data for inference of image processing and pattern recognition. For supporting full shuffling efficiently, Overlay-ML offers two read paths: a *fast read path* and a *slow read path*.

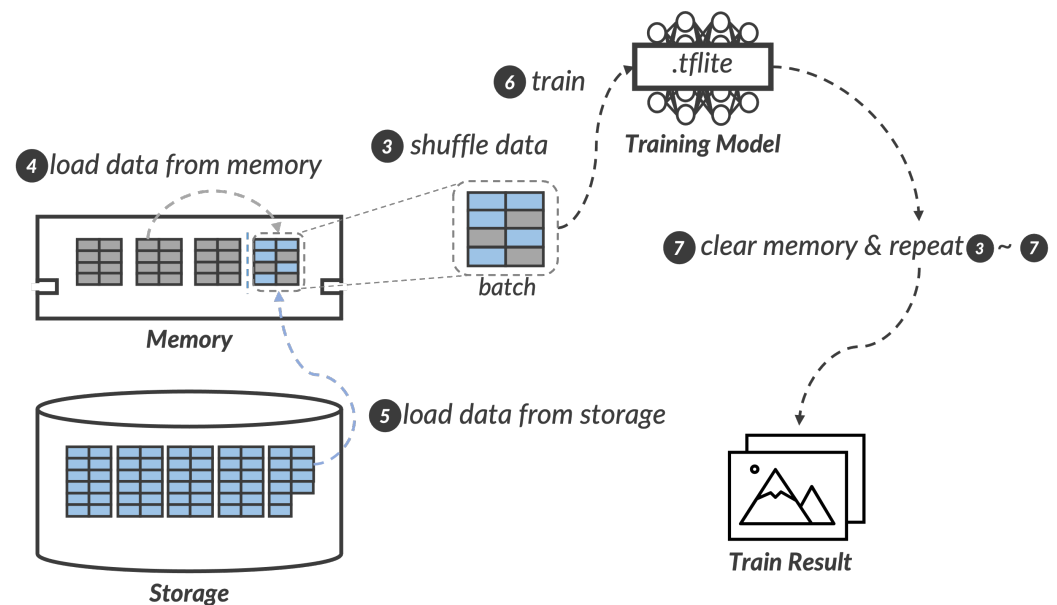


Figure 4. The overall flows of the overlay-shuffle-read operation in Overlay-ML.

Unlike the above write paths, the read path is determined according to the information of the mapping table that records the location of data. Overlay-ML first randomly selects data and finds the location of the corresponding data in the mapping table. If the location is mapped to the memory space, Overlay-ML triggers the fast read path to load data from the memory space (denoted by ④). Otherwise, it issues a read operation to the underlying storage device to load data via the slow read path (denoted by ⑤). After finishing the pre-processing step, Overlay-ML executes the training and validation steps based on tensors shuffled prior step (denoted by ⑥). Running ML model cannot be aware of where data is loaded since Overlay-ML provides transparency of data. Then, Overlay-ML reclaims unnecessary memory space that is not used in the future. It is repeated from the pre-processing to the validation step until all epochs end (denoted by ⑦). Finally, Overlay-ML clearly removes the mapping table and dedicated file corresponding to an application after terminating the application.

Note that the performance caused by slow read/write paths is poor because it is orders of magnitude slower than the read/write latency of memory. However, it is meaningful in

that they help to complete all training and validation steps belonging to the application using on-device ML without any unexpected termination.

4. Evaluation

In this section, we describe our experimental setup and evaluate Overlay-ML against the traditional on-device ML. Especially, we wish to answer the following two questions:

- Can Overlay-ML complete the training and validation for inference without unexpected termination on embedded devices?
- How does shuffling of Overlay-ML contribute to the final loss value?

4.1. Experimental Setup

All evaluations were performed on two state-of-the-art smartphones that are considered high-end embedded devices. Table 1 summarizes hardware specifications in detail.

Table 1. Specifications of two smartphones.

Name	Item	Specifications
Google Pixel 6 Pro	Chip	Google Tensor
	CPU (octa-core)	2×2.80 GHz Cortex-X1
		2×2.25 GHz Cortex-A76
		4×1.80 GHz Cortex-A55
	GPU	Mali-G78 MP20
	RAM	12 GB
Samsung Galaxy S20+	Storage	128 GB (UFS 3.1)
	OS	Android 12
	Chip	Qualcomm SM8250 Snapdragon 865 5G
	CPU (octa-core)	1×3.0 GHz Kryo 585
		3×2.42 GHz Kryo 585
		4×1.8 GHz Kryo 585
	GPU	Adreno 650
	RAM	8 GB
	Storage	256 GB (UFS 3.0)
	OS	Android 10

For evaluation, we first built up evaluation environments based on TensorFlow Lite [28] and implemented an application with the TFLite on-device personalization model that is a pre-trained MobileNetV2 model to recognize image datasets [15] (i.e., transfer learning). We also modified the implemented model to enable Overlay-ML features to run in a lightweight way on two different smartphones and trained our private datasets that contain around 3000 images with the pre-trained model. Since the memory allocated to the application is beyond the 512 MB limit after loading around 2100 images, the number of images in the private datasets is large enough.

- Baseline: this is the default object detection model that classifies and detects objects in a given image or a video stream [28].
- Overlay-ML (w/o Hybrid Shuffle): it follows of basic rules in Overlay-ML except for the shuffling step. In this model, it shuffles data in the main memory and data in the underlying storage device, separately.
- Overlay-ML: it enables all features of Overlay-ML based on the default object detection model.

To confirm how Overlay-ML works in memory pressure on high-end embedded devices, we evaluate the resistance of Overlay-ML by varying the number of images, which should simultaneously be accommodated and handled for performing image classification and recognition. We performed each experiment by repeating it five times to obtain a stable

and average value after rebooting devices; other applications never run except for system applications, such as phones, SMS, etc.

4.2. Overall Performance

We first present the overall performance of all models and describe how Overlay-ML survives in memory pressure.

Figures 5 and 6 show our evaluation results. As expected, Android OS terminated the baseline application around the point where loading images of about 2100. In other words, as shown in Figure 2, the baseline used the memory beyond the 512 MB limit. Therefore, we cannot present the performance results of the baseline after 2500 images in both. On the other hand, two applications with Overlay-ML features survived even though they exceeded the memory limitation (i.e., 512 MB) because of the key idea of Overlay-ML. The performance of applications with Overlay-ML features shows a similar performance to that of baseline at 1500 images. The reason is that all applications can service the training and validation procedures in memory space without any memory pressure; all data are stably serviced via fast write or read paths. Meanwhile, the elapsed time of two applications with Overlay-ML is higher than that of baseline at the number of 2000 images. The reason behind these results is that Overlay-ML triggers the operations of slow write and read paths when the total memory usage exceeds the pre-defined thresholds in Overlay-ML (i.e., 95% of the memory limitation). We leave some memory space to run the hybrid shuffling operations (i.e., 5% of the memory limitation). Finally, two applications with Overlay-ML show similar performance in all cases except on 2500 images in Figure 5 and 2000 images in Figure 6. We have analyzed the reasons for the difference and discovered that I/O interference occurs at the storage level. In other words, since two applications with Overlay-ML perform I/O operations via the slow write path, they can interfere with I/O operations issued by running applications or operating systems. We believe the performance difference is negligible because the loss value is more important on the device ML model.

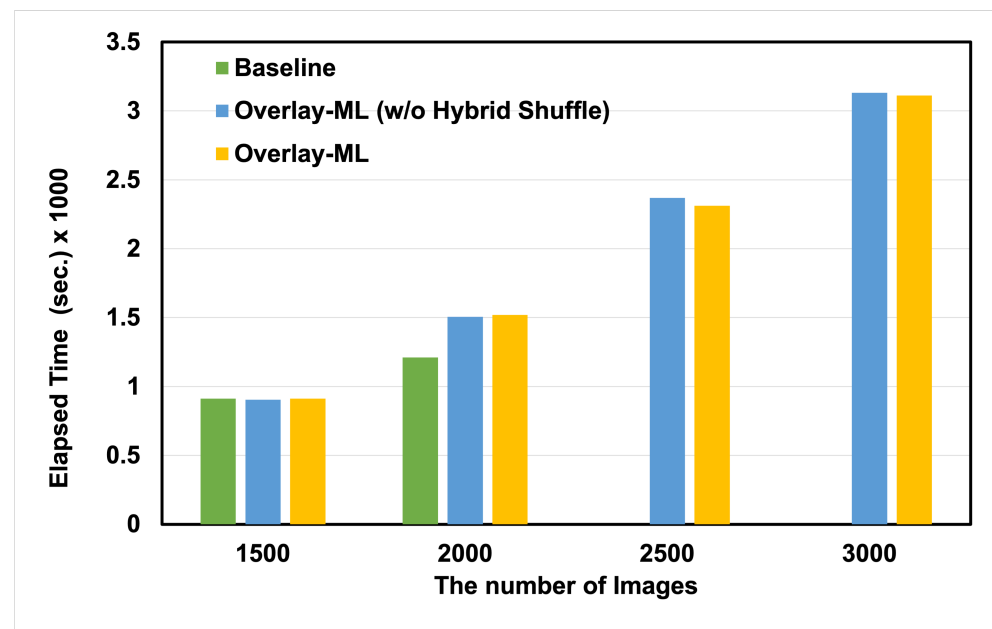


Figure 5. Performance results on Google Pixel 6 Pro.

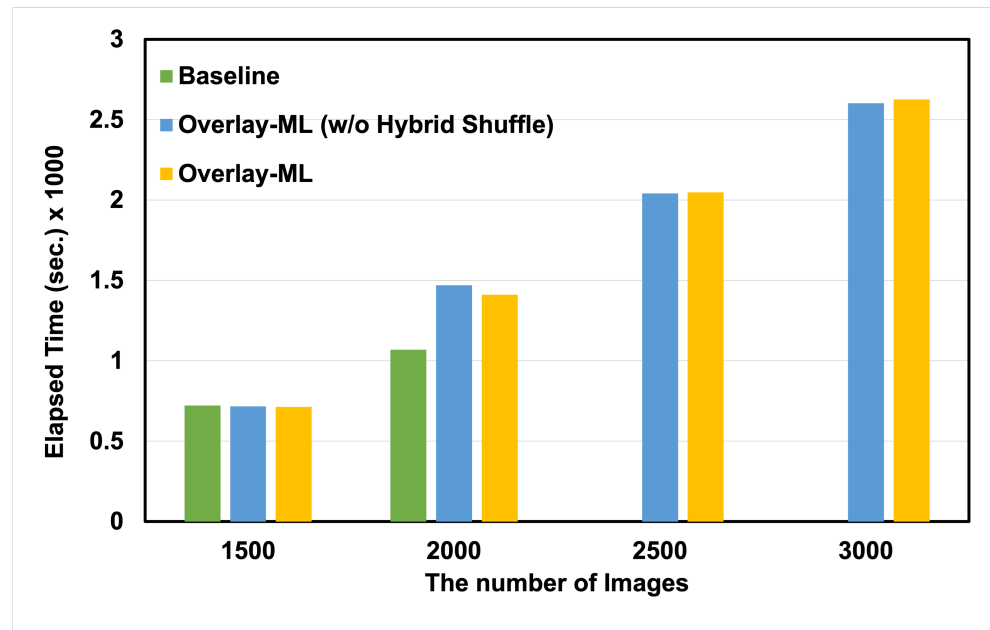


Figure 6. Performance results on Samsung Galaxy S20+.

4.3. Loss Value

Now, we describe the loss value of Overlay-ML. As shown in the previous section, the performance drop in Overlay-ML is unavoidable because it employs the operation of slow write and read paths. However, Overlay-ML never allows a drop in the loss value; it is one of the important parameters of the inference model. Therefore, we measured the loss value while running the application based on the epoch.

First, we show our evaluation results when the number of images is 2000 (see Figures 7 and 8). As we expected, all applications regardless of Overlay-ML features show good loss value (lower is better). This is understandable because all data in the main memory is shuffled evenly and randomly; it means there are no overfitting or bias issues.

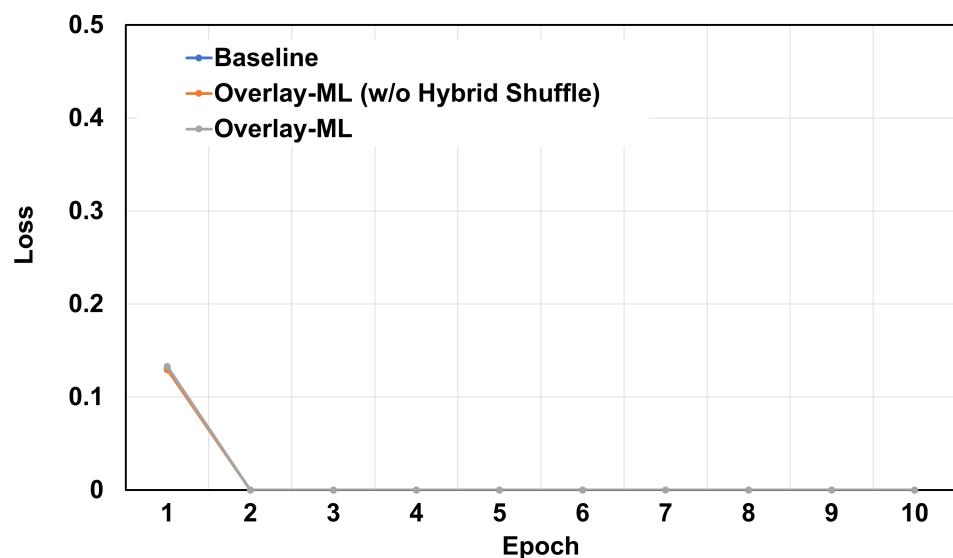


Figure 7. Loss on Google Pixel 6 Pro when the number of images is 2000.

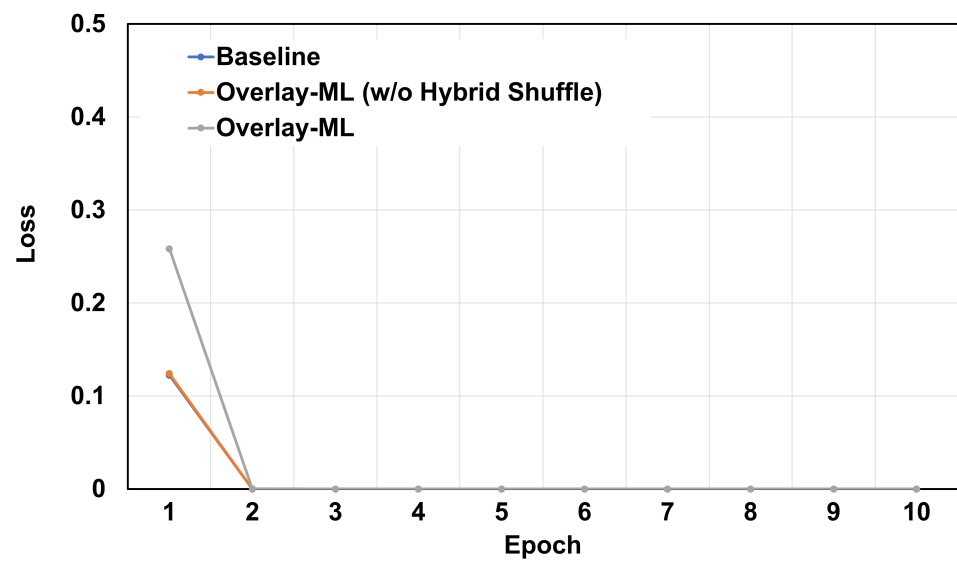


Figure 8. Loss on Samsung Galaxy S20+ when the number of images is 2000.

Now, we describe the evaluation results when the number of images is 3000. Figures 9 and 10 show the measured loss value while running the applications with Overlay-ML feature. Note that the baseline application was terminated as mentioned before. Unlike previous results (see Figures 7 and 8), Figures 9 and 10 show quite different loss values. Especially, the application with Overlay-ML (w/o hybrid shuffle) significantly increases the loss value because it suffers from overfitting or bias issues. On the other hand, the loss value of the application with Overlay-ML is similar to that of the previous evaluation (i.e., when the number of images is 2000). It is possible because of the hybrid shuffle feature of Overlay-ML. In other words, data in the main memory or the underlying storage device is fully mixed via the hybrid shuffle and it prevents the issues caused by data overfitting or bias.

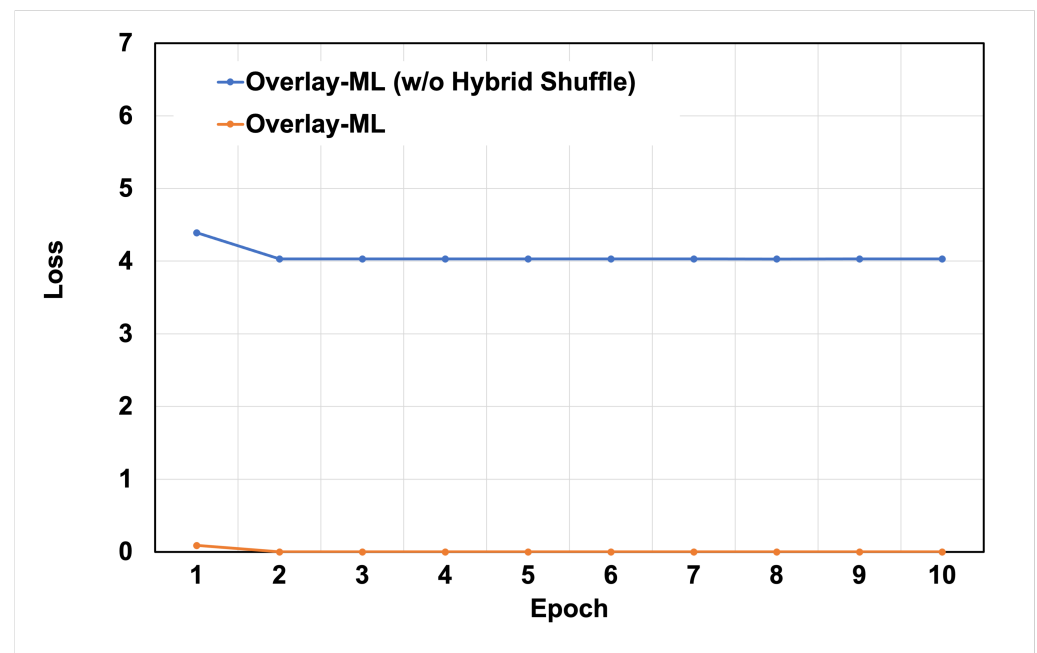


Figure 9. Loss on Google Pixel 6 Pro when the number of images is 3000.

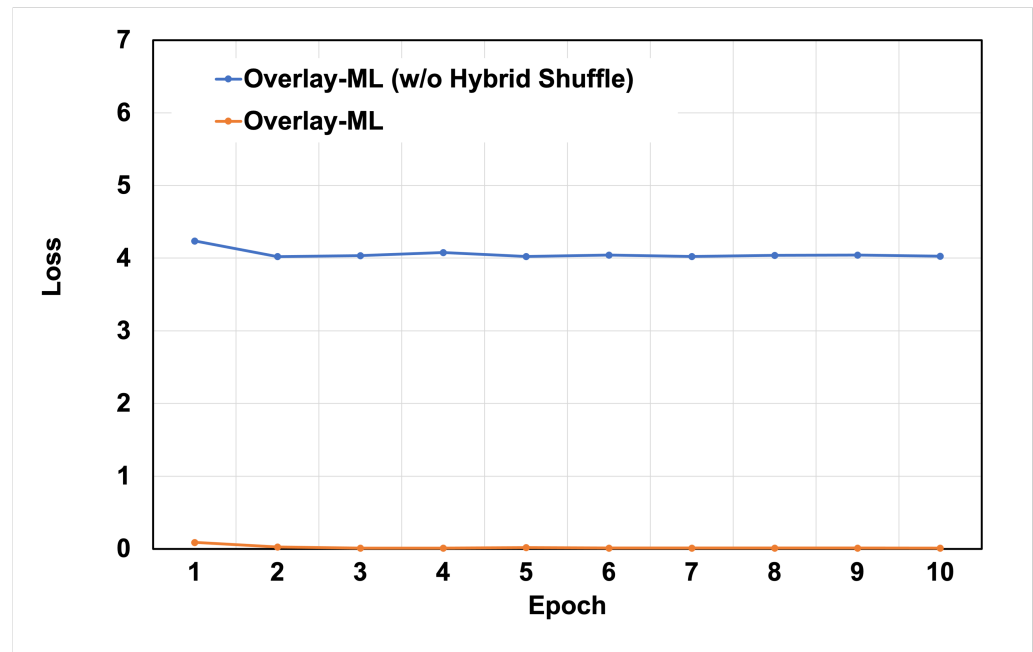


Figure 10. Loss on Samsung Galaxy S20+ when the number of images is 3000.

In summary, Overlay-ML supporting user-level application helps to finish the training and validation procedures in any memory pressure and guarantees good loss value that comes from the hybrid shuffle feature. However, it sometimes delays time to perform the operation for slow write and read paths. We believe this is unavoidable and it can be optimized by re-designing the logic of slow paths.

5. Conclusions

The traditional on-device ML framework has several inevitable challenges; (C1) hardware limitation because of shortage of memory space, (C2) functionality limitation due to the early stage of embedded software, and (C3) software limitation caused by the memory management policy of Android OS.

In this paper, we propose a novel data management policy on-device ML, Overlay-ML, that leverages the space extension feature to address the challenges C1 and C3, simultaneously. Our experimental results using the Overlay-ML prototype built on TensorFlow Lite clearly show that Overlay-ML prevents the unexpected termination issue caused by Android OS in memory pressure and completely finishes all training and validation procedures for image processing and pattern recognition. In addition, Overlay-ML guarantees good loss value by providing data transparency that is invisible outside of it. We believe many researchers and engineers using various sensing devices can enjoy the benefits of Overlay-ML in the real world since more hardware resource limitations are waiting to be solved in low-end embedded devices.

Meanwhile, there can be a wide variety of interference among running applications at the kernel level because resources (e.g., memory, CPU, and I/O stacks) are shared. Therefore, we further discover the interference issues by analyzing the application's operations on mobile devices, which would be an interesting future work.

Author Contributions: Conceptualization, C.K. and D.K.; methodology, D.K.; software, C.K.; validation, C.K.; formal analysis, D.K.; investigation, D.K.; resources, D.K.; data curation, D.K.; writing—original draft preparation, D.K.; writing—review and editing, D.K.; visualization, D.K.; supervision, D.K.; project administration, D.K.; funding acquisition, D.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2023-00251730).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ML	Machine learning
UAV	Unmanned aerial vehicle
ROV	Remotely operated vehicle
DL	Deep learning
TL	Transfer learning
OS	Operating system
IoT	Internet of things
LMKD	Low memory killer daemon

References

1. Thapa, A.; Horanont, T.; Neupane, B.; Aryal, J. Deep Learning for Remote Sensing Image Scene Classification: A Review and Meta-Analysis. *Remote Sens.* **2023**, *15*, 4804. [CrossRef]
2. Yao, Y.; Shi, Z.; Hu, H.; Li, J.; Wang, G.; Liu, L. GSDerainNet: A Deep Network Architecture Based on a Gaussian Shannon Filter for Single Image Deraining. *Remote Sens.* **2023**, *15*, 4825. [CrossRef]
3. Sharahi, H.J.; Acconcia, C.N.; Li, M.; Martel, A.; Hynynen, K. A Convolutional Neural Network for Beamforming and Image Reconstruction in Passive Cavitation Imaging. *Sensors* **2023**, *23*, 8760. [CrossRef] [PubMed]
4. Zhang, Z.; Huang, J.; Hei, G.; Wang, W. YOLO-IR-Free: An Improved Algorithm for Real-Time Detection of Vehicles in Infrared Images. *Sensors* **2023**, *23*, 8723. [CrossRef]
5. Image Sensor Market. Available online: <https://www.marketsandmarkets.com/Market-Reports/Image-Sensor-Semiconductor-Market-601.html> (accessed on 2 November 2023).
6. RISC-V. Available online: <https://riscv.org/> (accessed on 16 October 2023).
7. Raspberry Pi. Available online: <https://www.raspberrypi.com/> (accessed on 16 October 2023).
8. Wu, S.; Wang, Y.; Zhou, A.C.; Mao, R.; Shao, Z.; Li, T. Towards Cross-Platform Inference on Edge Devices with Emerging Neuromorphic Architecture. In Proceedings of the Design, Automation and Test in Europe Conference (DATE'19), Grenoble, France, 9–13 March 2019; pp. 806–811.
9. Guo, P.; Hu, B.; Hu, W. Automating DNN Model Porting for On-Device Inference at the Edge. In Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21), Boston, MA, USA, 12–14 April 2021; pp. 705–719.
10. He, X.; Wang, X.; Zhou, Z.; Wu, J.; Yang, Z.; Thiele, L. On-Device Deep Multi-Task Inference via Multi-Task Zipping. *IEEE Trans. Mob. Comput.* **2023**, *22*, 2878–2891. [CrossRef]
11. Lv, C.; Niu, C.; Gu, R.; Jiang, X.; Wang, Z.; Liu, B.; Wu, Z.; Yao, Q.; Huang, C.; Huang, P.; et al. Walle: An End-to-End, General-Purpose, and Large-Scale Production System for Device-Cloud Collaborative Machine Learning. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22), Carlsbad, CA, USA, 11–13 July 2022; pp. 249–265.
12. Tensorflow Lite. Available online: <https://www.tensorflow.org/lite> (accessed on 20 July 2022).
13. PyTorch Mobile. Available online: <https://pytorch.org/mobile/home> (accessed on 20 July 2021).
14. ML Kit. Available online: <https://developers.google.com/ml-kit?hl=en> (accessed on 15 August 2023).
15. Github:Tensorflow Lite Example. Available online: https://github.com/tensorflow/examples/tree/master/lite/examples/model_personalization (accessed on 20 July 2022).
16. Choi, J.; Kang, D. Overlapped Data Processing Scheme for Accelerating Training and Validation in Machine Learning. *IEEE Access* **2022**, *10*, 72015–72023. [CrossRef]
17. Tensorflow. Available online: <https://www.tensorflow.org/?hl=en> (accessed on 20 July 2021).
18. Abadi, M.; Paul, B.; Jianmin, C.; Zhifeng, C.; Andy, D.; Jeffrey, D.; Matthieu, D.; Sanja, y.G.; Geoffrey, I.; Michael, I.; et al. TensorFlow: A System for Large-Scale Machine Learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16), Savannah, GA, USA, 2–4 November 2016; pp. 265–283.
19. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 8026–8037.
20. Transfer Learning. Available online: https://en.wikipedia.org/wiki/Transfer_learning (accessed on 18 October 2023).
21. Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; Liu, P.J. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* **2020**, *21*, 5485–5551.

22. Best Embedded Operating Systems (OS) Reviews 2023. Available online: <https://www.gartner.com/reviews/market/embedded-operating-systems> (accessed on 18 October 2023).
23. Gutiérrez, J.; Villa-Medina, J.F.; Nieto-Garibay, A.; Ángel Porta-Gándara, M. Automated Irrigation System Using a Wireless Sensor Network and GPRS Module. *IEEE Trans. Instrum. Meas.* **2014**, *63*, 166–176. [CrossRef]
24. Learning Plan Resources for Azure IoT Hub. Available online: <https://microsoft.github.io/PartnerResources/azure/iot/iot-hub> (accessed on 18 October 2023).
25. Son, S.; Lee, S.Y.; Jin, Y.; Bae, J.; Jeong, J.; Ham, T.J.; Lee, J.W.; Yoon, H. ASAP: Fast Mobile Application Switch via Adaptive Prepaging. In Proceedings of the 2022 USENIX Annual Technical Conference (ATC'21), Boston, MA, USA, 14–16 July 2021; pp. 117–130.
26. Lim, G.; Kang, D.; Ham, M.; Eom, Y.I. SWAM: Revisiting Swap and OOMK for Improving Application Responsiveness on Mobile Devices. In Proceedings of the 29th Annual International Conference On Mobile Computing And Networking (MobiCom'23), Madrid, Spain, 2–6 October 2023; pp. 1–15.
27. Android:largeHeap. Available online: <https://developer.android.com/guide/topics/manifest/application-element#largeHeap> (accessed on 20 July 2021).
28. Object Detection. Available online: https://www.tensorflow.org/lite/examples/object_detection/overview (accessed on 21 October 2023).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.