Characterizing the Performance of Deep Learning Workloads on Accelerated Edge Computing Devices

Prashanthi S.K*, Sai Anuroop Kesanapalli, Aakash Khochare* and Yogesh Simmhan
Department of Computational and Data Sciences
Indian Institute of Science
Bangalore 560012 INDIA
Email: prashanthis@iisc.ac.in, saiak@iisc.ac.in, aakhochare@iisc.ac.in, simmhan@cds.iisc.ac.in

***Abstract*—Deep Learning (DL) models have had a significant impact on domains like autonomous vehicles, urban safety and Internet of Things (IoT) by enabling low-latency inferencing on edge computing devices, close to the data source. With the massive growth in sensor and camera data from such domains, the need to maintain freshness of models through retraining, and the heightened attention to privacy, *training* of DL models on GPU-accelerated low-power edges like NVIDIA Jetson through techniques like federated learning is gaining importance. Such training is resource-intensive and can stress the capacity of an edge's resources like GPU, CPU, memory and storage. While previous studies have profiled the resource usage and identified bottlenecks of ML workloads on accelerated Cloud VMs and server, such a characterization has been absent for edge devices whose field-deployments are rapidly increasing. In this work, we closely examine a ML model training on the Nvidia Jetson Xavier AGX and Xavier NX. We vary several training and device parameters such as dataset size, I/O threads and storage device, and measure the CPU and GPU compute time and utilization, fetch stalls, and end-to-end time to understand the bottlenecks in the training pipeline. Our analysis identifies several interesting insights on the effect of storage medium and caching on the training time in the edge.**

I. Introduction

Deep Learning (DL) models find application in a variety of contemporary domains such as Autonomous Vehicles [1], Smart cities [2] and Healthcare [3]. They help inference over video data for drones navigation, allow safety cameras to identify suspicious activities, and examine images for medical diagnostics. Fast inferencing close to the data source is enabled through a growing class of accelerated edge devices such as NVIDIA Jetson and Google Coral which host low-end GPUs and TPUs along with ARM CPUs and a compact form-factor to offer a superior performance-to-energy ratio. For instance, the Nvidia Xavier AGX development kit has a 512-core Volta GPU, an 8 core ARM CPU and 32GB of LPDDR4x memory, operates within 65W of power and costs US$700.

Recently, there has been a push toward *training* DL models on the edge. This is driven by the massive growth in data collected from edge devices, the need to refresh the models periodically, the bandwidth constraints in moving all this data to Cloud data centers for training, and a heightened attention to privacy by retaining data on the edge. This has led to techniques like federated and geo-Distributed learning [4] that train local models on a device and aggregate the models on a central server to build a global model. The local training phase iteratively optimizes the model parameters with the objective of minimizing a loss function.

This proliferation of DL model training has led to an increasing interest in scrutinizing the systems' characteristics of such workloads to help identify bottlenecks, optimize the training parameters and even to choose the machine configuration. However, this has been largely limited to evaluating their performance on GPU-accelerated Cloud VMs and servers [5], [6], with minimal investigations of edge devices. Edge devices have several unique characteristics – a limited sized RAM, often shared between GPU and CPU, and diverse storage media such as SD cards, EMMC and NVME. So, understanding the bottlenecks unique to edge devices is essential to design frameworks and systems which optimally utilize the constrained hardware resources. This can also help in making informed design choices on selecting the configuration of edge devices that are tailored for DL workloads.

In this paper, we conduct detailed experiments using the current-generation Nvidia Jetson Xavier AGX and NX edge platforms and representative DL models and datasets to make the following contributions:

1) We understand the effect of parallel data fetch and pipelining of data access and pipelining on overall training time
2) We characterize the impact of the storage medium and disk caching on data fetch stalls

Our analysis is used to draw insights on different device tuning parameters and device configurations that can improve the DL training time.

II. Experiment Methodology

A. Hardware Platforms

1) Jetson Devices: We perform our experiments on two Nvidia Jetson edge device developer kits: AGX Xavier [7] and Xavier NX [8]. These are detailed below. The CUDA version used is 10.2. We use Jetpack version 4.5.1 with L4T(Linux for Tegra) version 32.5.1 and kernel version 4.9.201-tegra

2) Storage Media: These edge devices offer connectivity to different storage devices. Since training can be I/O intensive, we perform experiments on three different storage media –

* Student Author

Table I: Default Device Specifications of Nvidia Jetsons

Feature	AGX	NX
CPU Cores	8	6
CPU Clockspeed (MHz)	2188	1500
GPU Clockspeed (MHz)	677	800
GPU Architecture	Volta	Volta
RAM (GB)	32	8
Power Budget (W)	15	10

Figure 1: DL Training pipeline stages on PyTorch

SSD over a NVMe/PCIe interface, *HDD over a USB interface*, and *SD card*.

B. Software Platform

The ML training pipeline has three stages – *fetch, pre-process* and *compute*. The data is first fetched from the storage device to the main memory in batches. Then the pre-processing operations such as crop/resize are performed on the CPU. Finally, the actual training over the data is done on the GPU in the compute stage. Training is done iteratively over many epochs till convergence. In each epoch, we load data in batches and each batch passes through the above pipeline. The model weights are updated at the end of every batch. The subset of the data used in each batch is randomized across different epochs. As we will see, this limits data locality and affects caching. Thus, the training pipeline involves multiple resources – storage, memory, CPU and GPU – all of which must be carefully tuned, especially on a constrained edge device, to avoid bottlenecks and reduce overall training time.

We use the PyTorch DL framework [9] for training the model, and the PyTorch Dataloader [10] to fetch and pre-process data. The dataloader performs the data fetch from disk followed by pre-process sequentially within a worker. We use the `num_workers` flag to vary the number of processes used to perform the fetch/pre-process. When `num_workers=0`, a single process performs fetch, pre-process *and* compute sequentially; this prevents pipelining. When `num_workers` $\geq$ `1`, Pytorch spins up that many processes for fetch/pre-process, each operating on a different batch of data in parallel, and a separate process invokes the GPU compute on each pre-processed batch sequentially. This forms two-stage pipeline of fetch/pre-process followed by compute.

C. DL Model and Dataset

1) Dataset: We use images from the Google Landmarks Dataset [11] for training. By default, we use the standard $23k$ version of the GLD dataset (GLD-23k), which has $\approx 23,080$ images with a total size of 2.8 GB. Each image file has a resolution of 800×600 with an average size of 100 kB.

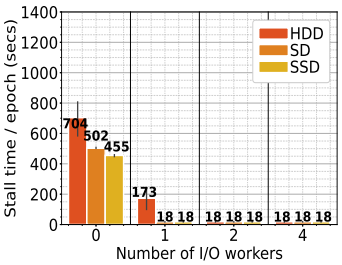


Figure 2: Stall Time | NX

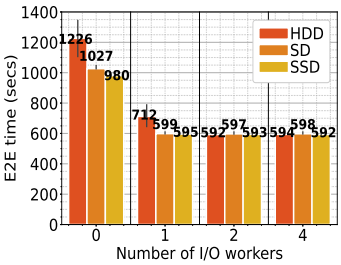


Figure 3: E2E Time | NX

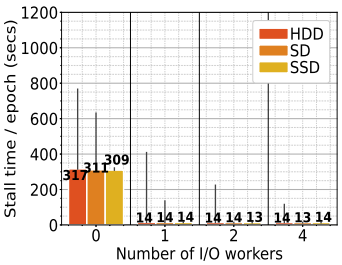


Figure 4: Stall Time | AGX

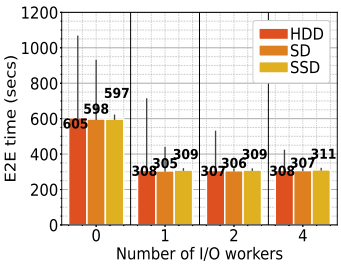


Figure 5: E2E Time | AGX

2) Model: We use MobileNetV3-Large [12] image analysis model for training. It is a lightweight model intended for mobile devices, auto-tuned using hardware-aware Network Architecture Search (NAS). The network comprises of conv2d layer, several bottleneck blocks followed by a pooling layer, culminating in two conv2d layers with no batch normalization. These suite of layers are drawn from its previous versions - MobileNetV1 [13], MobileNetV2 [14] and MnasNet [15], which includes squeeze and excitation into the bottleneck structure, and are upgraded with modified swish non-linearities.

3) Training Parameters: We use a batch size of 16 images unless mentioned otherwise. The learning rate and momentum are set to 0.01 and 0.9, respectively. We use Stochastic Gradient Descent (SGD) as the optimizer, and use cross-entropy as the loss function. We run all our experiments for 3 epochs of training since that is adequate to understand and generalize the performance behavior. Also, for some configurations, each epoch runs for over 93 minutes. We do not include a testing phase in our experiments as we are not training till convergence.

III. Results and Analysis

Metrics. We use a variety of Linux utilities to measure various system resources. *CPU, GPU* and *RAM utilisation* are measured using the `tegrastats` [16] utility from Nvidia every 1 second. I/O metrics such as *read IOPS* and *bytes read per second (throughput)* are measured using `iostat` [17]. The fraction of the dataset that is present in the Linux (in-memory) disk cache is measured using `vmtouch` [18].

Additionally, we measure the fetch stall time and the GPU compute time for every batch. *Fetch stall time* is the *visible* time taken to fetch and pre-process data and does not overlap with the GPU compute time, i.e., $\max$(fetch time +

pre-process time − GPU compute time, 0). *GPU compute time* is the time taken by the batch to execute on the GPU and includes the forward and backward passes of training. We measure these times using the `torch.cuda.event` with the `synchronize` option so that time captured is accurate. We sum up the fetch stall and GPU compute times over all batches to obtain the per-epoch fetch stall time and compute time, which we report in our analyses. We also measure the *End-to-End (E2E) time* to process all batches of each epoch, and this includes the fetch stall time, compute time and also any other framework overheads. The power mode for the AGX is MAXN (i.e., there is no power budget constraint and it can consume the peak rated 65W), and for NX it is 15W and using 6 CPU cores. In all our plots, we report the median fetch stall, compute time and E2E epoch times.

A. *Pipelining reduces fetch stalls, and hence E2E time.*

Fetching data and pre-processing involve the CPU, while compute is entirely done on the GPU. If these are executed sequentially, the GPU remains idle while it waits for the CPU to finish fetching and pre-processing data. By pipelining the fetch/pre-process with compute, the stall time reduces from the entire fetch/pre-process time to just (fetch/pre-process - compute). Pipelining is enabled by increasing the number of workers in the PyTorch Dataloader to ≥ 1.

Figs. 2 and 3 report the median fetch and pre-process *stall time*, i.e., the duration for which the compute process was idle waiting for a batch to be ready, and the *E2E time* as the number of workers increases from 0 (no pipelining) to 4 (4 parallel fetch/pre-process workers pipelining to 1 compute process), for the NX device. Figs. 4 and 5 show the same for AGX. We focus on the increase from 0 to 1 worker here.

The effect of pipelining can be seen from the reduction in cumulative fetch stalls per epoch and end to end times. For the NX, when the number of workers increases from 0 to 1, the fetch stall reduces from (704, 502, 455 $secs$) to (173, 18.4, 18.6 secs) for the HDD, SD, and SSD, respectively (Fig. 2). Notably, just introducing pipelining by going from 0 to 1 worker offers a significant improvement, and decreases the stall time by $3.1\times$ for the HDD, $26.2\times$ for the SD card, and $23.4\times$ for the SSD. We see a consequent decrease in the E2E epoch time from (1226, 1027, 980 $secs$) to (713, 599, 595 $secs$) for the HDD, SD and SSD respectively (Fig. 3).

For the AGX, in Fig. 4, by introducing pipelining, the fetch stalls reduce from (317, 311, 309 $secs$) to (14.7, 14.6, 14.5 $secs$) for the HDD, SD card and SSD – a substantial decrease in time of ($20.56\times$, $20.3\times$, $20.31\times$). Correspondingly, in Fig. 5, the E2E time drops from (605, 598, 597 $secs$) at 0 workers to (308, 305, 309 $secs$) with 1 worker.

All the storage media see an equal decrease in time on the AGX because the dataset is entirely present in the in-memory disk cache for the second and third epochs, and fetch stalls occur only in the first epoch where accesses go to the disk. However, NX has more limited RAM and the entire data does not fit in cache, causing repeated fetches from disk for each epoch. We also observe a lower GPU utilization with 0 workers, with median values of 0%, 19% and 61% for HDD, SD and SSD, respectively, on the NX, which increases to 99% with pipelining.

B. *Parallelization of fetch may give further improvements depending on the storage medium and device.*

Number of workers can be increased beyond 1 to fetch and pre-process data in parallel across batches. This could potentially reduce the fetch stall time and provide an added benefit over just pipelining with 1 worker. But realizing this benefit depends on the storage medium and the edge device.

1) Parallel fetch has greater benefits for a slower storage medium such as HDD: As seen from Fig. 2 for NX on the HDD, increasing number of workers from 1 to 2 reduces the fetch stall time from 173.8 $secs$ to 18.9 $secs$, a reduction of $8.19\times$. However, for faster devices such as the SD card and SSD, going from 1 worker to 2 has no additional benefits since the fetch/pre-process of a single batch is fast enough to keep the NX's GPU compute occupied in processing that batch through pipelining, and the compute fully hides the fetch time.

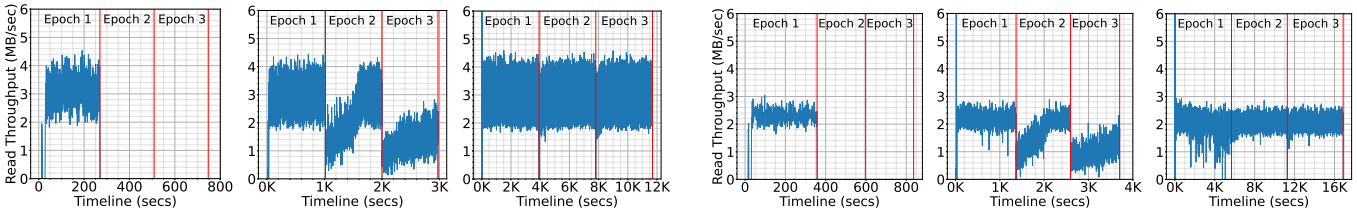
2) Parallel fetch has greater benefits for a device with a faster GPU relative to the disk: As the GPU on a device gets faster, the gap between CPU compute time and the fetch/pre-processing time widens, dominated by the disk I/O time. As a result, having 1 worker may not be sufficient to mitigate the fetch stall time. Increasing the number of workers has benefits here. For the AGX on the HDD and SD card, 1 worker cannot fully fill the input pipeline of the GPU compute. For the SD card, the fetch stall time in the first epoch reduces from 134 $secs$ with 1 worker to 12.3 $secs$ with 2 workers. For the HDD, the fetch stall time in the first epoch reduces from 408.2 $secs$ with 1 worker to 223.9 $secs$ with 2 workers and reduces further to 115.4 $secs$ with 4 workers. We report only first epoch times here because the dataset is entirely cached in RAM, and only the first epoch times are reflective of disk access.

C. *A storage medium whose fetch time can be hidden by the compute time is sufficient. Having a medium faster than that offers no additional benefits.*

In Fig. 2, using parallel fetch with 2 workers on the NX, the fetch stall time of the HDD is almost entirely hidden by the compute time. Using an SSD/SD card which have faster I/O speed does not give us any additional benefits. All three storage media have comparable E2E epoch times of $\approx$ 590 $secs$ with 2 workers (Fig. 3). On the AGX using 2 workers (Fig. 4), the SD card has fetch stalls comparable to the SSD (14 and 13 $secs$ respectively) and is sufficient.

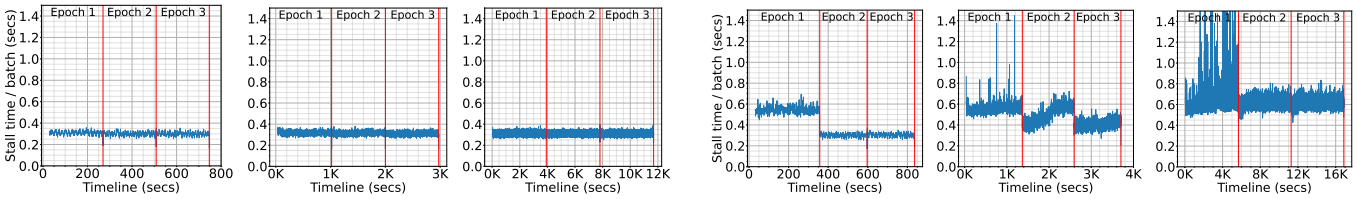
D. *Having a large enough RAM and consequently disk cache can compensate for a slower storage medium.*

The Linux disk cache uses the available free RAM to retain recently fetched files in-memory. If *all the dataset* required for training in an epoch is already present in the cache, then the accesses goes to RAM and not the disk beyond the first epoch



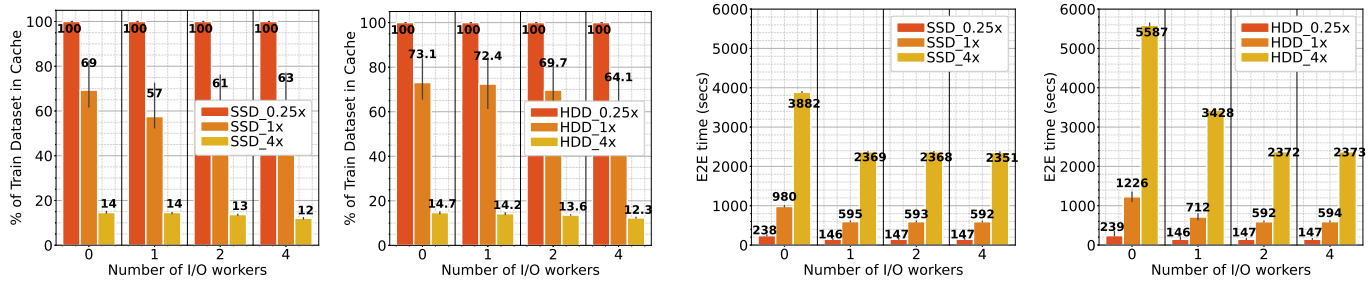
(a) SSD | 0.25x (b) SSD | 1x (c) SSD | 4x (d) HDD | 0.25x (e) HDD | 1x (f) HDD | 4x

Figure 6: IO Reads (Size) | NX



(a) SSD | 0.25x (b) SSD | 1x (c) SSD | 4x (d) HDD | 0.25x (e) HDD | 1x (f) HDD | 4x

Figure 7: Fetch Stall Time | NX



(a) SSD (b) HDD (c) SSD (d) HDD

Figure 8: % of Train Dataset in Cache and E2E Time | NX

when the cache is initially populated. Since training typically runs over a number of epochs, the penalty for loading from disk in the first epoch is amortized over the subsequent epochs where accesses is completely from RAM. This is observed for the AGX, which has 32 GB of RAM and is sufficient to fit the 2.8 GB of training data fully in disk cache. So using a slower storage medium such as HDD has no penalty beyond the first epoch. E.g., in the first epoch, we observe E2E times of around (1060, 925, 620 $secs$) for (HDD, SD card, SSD) with 0 workers on AGX. But this drops to around 300 $secs$ in epochs 2 and 3 for all 3 storage media.

*E. **Benefits of caching are not seen when I/O pressure on the storage medium is low.***

We synthetically scale the dataset to $0.25\times$ and $4\times$ of the original 23k images in order to quantify the imapact of various dataset sizes. Figs 6 and 7 show the time series plots for IO Reads and Fetch Stall Time on the Jetson NX. The red vertical lines indicate epoch boundaries. In Fig. 8a we see that smaller fractions of the data are cached in memory as the size of the dataset increases. Consequently, the number of reads going to disk also increases in epochs 2 and 3 as seen from Figures 6a to 6c. (Epoch 1 starts with a cold cache, so all reads go to disk). However, the fetch stalls time for the SSD does not vary with dataset size as seen from Figs 7a to 7c. This is because the I/O pressure is low enough that the SSD appears as fast as the RAM, although it is an order of magnitude slower. Dataset caching has no impact on the fetch stall time at low I/O pressure. From Fig. 8c, we can see a perfectly linear increase in E2E epoch time with increase in dataset size. For example, in going from $1\times$ to $4\times$ the data, E2E epoch time goes from 600 to 2400 $secs$ (exactly $4\times$).

*F. Corollary: **With a slower medium such as the HDD, benefits of caching are evident.***

We repeat the same dataset scaling experiment on the HDD. Although the fraction of the dataset cached and the read patterns remain the same (Fig. 8b and Figs. 6d to 6f), the fetch stalls are different for epoch 1 (disk reads) versus epochs 2 and 3 (disk + cache reads). This can be seen in Fig. 7d, where the disk fetch stall in the first epoch is around 550 $millisecs$ and the subsequent epochs see fetch stalls of 300 $millisecs$ ($\sim 2\times$

lower). The benefits of caching are evident in the E2E time as well. We see from Fig 8d that increasing the dataset size from $1\times$ to $4\times$ with 0 workers results in E2E time increasing from 1200 to 5600 $secs$ ($4.67\times$). The overhead is due to many more reads.

IV. Related Work

A. Benchmarking DNN Training on Server GPUs

MLPerf [19] is a joint effort by the industry and academia that attempts to provide a uniform framework for quantifying the performance of ML Hardware and Systems. The Benchmark Suite spans a number of application domains and datasets, and prescribe a quality threshold that must be met by any implementation. While such a suite is essential for measuring the overall impact of systems or optimizations on training, it does not measure low level system metrics like the IO reads, caching, etc. MLPerf also lacks a Training suite for Edge devices.

[5] characterizes the data pipeline and how it affects training time on Desktop GPUs. It also analyzes the effects of the OS page cache on data access. However, it only considers server grade GPUs which are much more powerful and have more RAM when compared to edge devices.

B. Optimized data access for Deep Learning Training

[5] proposes a modified caching mechanism that minimises I/O caused by thrashing. Once the page cache is full, all further accesses are sent to disk without evicting existing data in the cache. Further, it proposes a partitioned caching mechanism that benefits distributed training. Quiver [20] proposes a caching strategy based on *substitutability*. Accesses that cause a miss in the cache are substituted with other data that are present in the cache without interfering with training requirements of randomness and single access per epoch.

C. Edge device characterization

DeepEdgeBench [21] is a DNN Benchmark that compares the performance of edge devices with respect to model inferencing. They report the inference time and power consumption for MobileNetv2 on edge devices such as Nvidia Jetson Nano, Google Coral Board and Raspberry Pi 4. Our work focuses on training instead of inference and also provides detailed insights into the behaviour of storage mediums on the edge class devices. Flower [22] is an open source Federated Learning (FL) framework that supports heterogeneous environments including mobile and edge devices, and scales to a large number of distributed clients. Their paper presents the results of deploying Flower on Android devices in the Amazon AWS Device Farm and on Nvidia Jetson TX2 edge accelerator.

V. Conclusion

In this paper, we have closely examined the system characteristics of ML Model training on Nvidia Jetson Xavier AGX and Xavier NX platforms. We provide a detailed analysis of the role the storage subsystem plays in model training on edge devices. We demonstrated the significance of pipelining and parallelism in avoiding data fetch stalls. Further, we analysed the role played by the Linux cache in avoiding fetch stalls on multiple dataset sizes. As future work, we plan on performing similar experiments for a broader set of DNN models. The edge devices also offer several power modes and it would be interesting to explore the tradeoff between the power budgets and the training performance. Early experiments show that using faster and fewer CPU cores (equal to the optimal number of workers) has performance benefits. Also, initial experiments show that low power modes offer significant energy savings at a small performance penalty.

References

[1] S. Kuutti *et al.*, "A survey of deep learning applications to autonomous vehicle control," *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[2] Q. Chen *et al.*, "A survey on an emerging area: Deep learning for smart city data," *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2019.

[3] D. Kollias *et al.*, "Deep neural architectures for prediction in healthcare," *Complex & Intelligent Systems*, 2018.

[4] Z. Zhou *et al.*, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, 2019.

[5] J. Mohan *et al.*, "Analyzing and mitigating data stalls in dnn training," *Proceedings of the VLDB Endowment*, 2021.

[6] Y. Wang *et al.*, "Benchmarking tpu, gpu, and cpu platforms for deep learning," *arXiv preprint arXiv:1907.10701*, 2019.

[7] Nvidia., "Jetson agx xavier developer kit," ftp://https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit, 2021.

[8] Nvidia, "Jetson nx xavier developer kit," ftp://https://developer.nvidia.com/embedded/jetson-xavier-nx, 2021.

[9] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, vol. 32, 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[10] pytorch, "Torch.utils.data," ftp://https://pytorch.org/docs/stable/data.html, 2021.

[11] T. Weyand *et al.*, "Google landmarks dataset v2-a large-scale benchmark for instance-level recognition and retrieval," in *IEEE/CVF conference on computer vision and pattern recognition*, 2020.

[12] A. Howard *et al.*, "Searching for mobilenetv3," in *IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE.

[13] 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.

[14] A. Howard, A. Zhmoginov, L.-C. Chen, M. Sandler, and M. Zhu, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," 2018.

[15] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 285–300.

[16] Nvidia, "tegrastats," ftp://https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide/AppendixTegraStats.html, 2021.

[17] man page, "iostat," ftp://https://man7.org/linux/man-pages/man1/iostat.1.html, 2021.

[18] man pages, "vmtouch," ftp://https://linux.die.net/man/8/vmtouch, 2021.

[19] P. Mattson *et al.*, "Mlperf training benchmark," vol. 2, 2020, pp. 336–349. [Online]. Available: https://proceedings.mlsys.org/paper/2020/file/02522a2b2726fb0a03bb19f2d8d9524d-Paper.pdf

[20] A. Kumar *et al.*, "Quiver: An informed storage cache for deep learning," in *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, 2020.

[21] S. Baller *et al.*, "Deepedgebench: Benchmarking deep neural networks on edge devices," in *IEEE International Conference on Cloud Engineering*, 2021.

[22] D. J. Beutel *et al.*, "Flower: A friendly federated learning research framework," *arXiv preprint arXiv:2007.14390*, 2020.