




Article

# Check-QZP: A Lightweight Checkpoint Mechanism for Deep Learning Frameworks

Sangheon Lee<sup>1</sup>, Gyupin Moon<sup>1</sup>, Chanyong Lee<sup>1</sup> , Hyunwoo Kim<sup>1</sup>, Donghyeok An<sup>2,\*</sup>  and Donghyun Kang<sup>1,\*</sup> 

<sup>1</sup> Department of Computer Engineering, College of IT Convergence, Gachon University, Seongnam-si 13120, Republic of Korea; tkdgjs0213@gachon.ac.kr (S.L.); kurizda@gachon.ac.kr (G.M.); bb9845@gachon.ac.kr (C.L.); kimh0425@gachon.ac.kr (H.K.)

<sup>2</sup> Department of Computer Engineering, Changwon National University, Changwon-si 51140, Republic of Korea

\* Correspondence: donghyeokan@changwon.ac.kr (D.A.); donghyun@gachon.ac.kr (D.K.)

**Abstract:** In deep learning (DL) frameworks, a checkpoint operation is widely used to store intermediate variable values (e.g., weights, biases, and gradients) on storage media. This operation helps to reduce the recovery time of running a machine learning (ML) model after sudden power failures or random crashes. However, the checkpoint operation can stall the overall training step of the running model and waste expensive hardware resources by leaving the GPU in idle sleep during the checkpoint operation. In addition, the completion time of the checkpoint operation is unpredictable in cloud server environments (e.g., AWS and Azure) because excessive I/O operations issued by other running applications interfere with the checkpoint operations in the storage stacks. To efficiently address the above two problems, we carefully designed Check-QZP, which reduces the amount of data required for checkpoint operations and parallelizes executions on the CPU and GPU by understanding the internal behaviors of the training step. For the evaluation, we implemented Check-QZP and compared it with the traditional approach in real-world multi-tenant scenarios. In the evaluation, Check-QZP outperformed the baseline in all cases in terms of the overall checkpoint time and the amount of data generated by the checkpoint operations, reducing them by up to 87.5% and 99.8%, respectively. In addition, Check-QZP achieved superior training speeds compared to the baseline.



**Citation:** Lee, S.; Moon, G.; Lee, C.; Kim, H.; An, D.; Kang, D. Check-QZP: A Lightweight Checkpoint Mechanism for Deep Learning Frameworks. *Appl. Sci.* **2024**, *14*, 8848. <https://doi.org/10.3390/app14198848>

Academic Editors: Francesco Zirilli and Andrew Teoh Beng Jin

Received: 28 August 2024  
Revised: 20 September 2024  
Accepted: 27 September 2024  
Published: 1 October 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/>).

**Keywords:** deep learning; checkpoint; quantization; parallelism; compression; performance and storage capacity

## 1. Introduction

Today, deep learning (DL) is emphasized even more as its technologies cover a wide spectrum of applications [1,2]. Many researchers have carried out studies that enable DL technologies to upgrade legacy services or provide new applications [3–8]. For example, some researchers have focused on autonomous vehicles or unmanned aerial vehicles (UAVs), incorporating DL technology to offer more intelligent services to clients [9–13]. Other researchers have enhanced custom-fit video recommendation systems using DL technology [14–16]. This research trend leads to excessive I/O operations for the learning and inference processes and an increase in data centers to store the vast amounts of data generated by applications or various sensors [17–19]. Unfortunately, the resources (e.g., CPU, GPU, memory, network, and storage devices) in the data centers must be shared among many applications in multi-tenant cloud environments [20,21]. Therefore, applications with AI engines have to compete for limited resources in the data centers, resulting in interference across applications in terms of CPU/GPU preemption, memory allocation, and storage I/Os.

Meanwhile, most platforms supporting AI engines sometimes write the bulk of their data to the underlying storage devices (e.g., HDDs or SSDs) while running the learning

process [22,23]. This is because the AI platforms have to store on-the-fly data that are generated during the training process to avoid losing the data in the event of a sudden power failure (i.e., *checkpoint* operation) [6,7]. In other words, after unexpected system crashes, the training process can be resumed using the data belonging to the last version of the checkpoint operation instead of performing the first step again. However, the data stored in the storage media are never reused unless the system crashes and are unavailable after running the next training epoch.

In addition, the checkpoint operation to store the data can negatively affect a series of training processes, as it must wait for the write operations to finish; it may write more than several GB of data [6,7,24]. Unfortunately, the latency of the write operations can be delayed when I/O interference unintentionally occurs due to excessive I/O operations from other running applications; we call this situation a *checkpoint delay* due to I/O conflict [24,25]. In recent years, some studies have focused on designing the checkpoint operation to address this delay in terms of I/O bandwidth and recovery time [6,7,26,27]. For example, the authors of Check-N-Run [7] adopted a quantization and parallelism mechanism into the checkpoint operation to deal with storage capacity and write bandwidth. The authors of CheckFreq [6] proposed a lightweight checkpointing framework for DNN training to speed up recovery time in the event of a sudden power failure. Unfortunately, previous studies do not consider the excessive I/O conflicts in cloud environments, where the latency of I/O operations from the checkpoint operations is limited because of interference from other running multi-tenant applications.

In this paper, we propose Check-QZP, which efficiently addresses the *checkpoint delay* issue by reducing the total amount of checkpointed data under excessive I/O conflict scenarios. The contributions of this paper can be summarized as follows:

- In order to understand how the training process works, we briefly introduce the internal procedure of the training process. Then, we describe why a *checkpoint delay* occurs and what kinds of problems can arise.
- We propose Check-QZP, which consists of three features—quantization, parallelism, and compression—to efficiently optimize overall performance and storage space.
- We implement Check-QZP at the framework level and compare it with the default mechanism in multi-tenant scenarios where the ML model performs training and inference under high I/O pressure from the excessive FIO [28] benchmark.

The rest of this paper is organized as follows. Section 2 introduces the background and related works to better understand our work. Section 3 explains how Check-QZP works and details its benefits. Finally, Section 4 presents our evaluation results, and Section 5 concludes this paper.

## 2. Background and Related Works

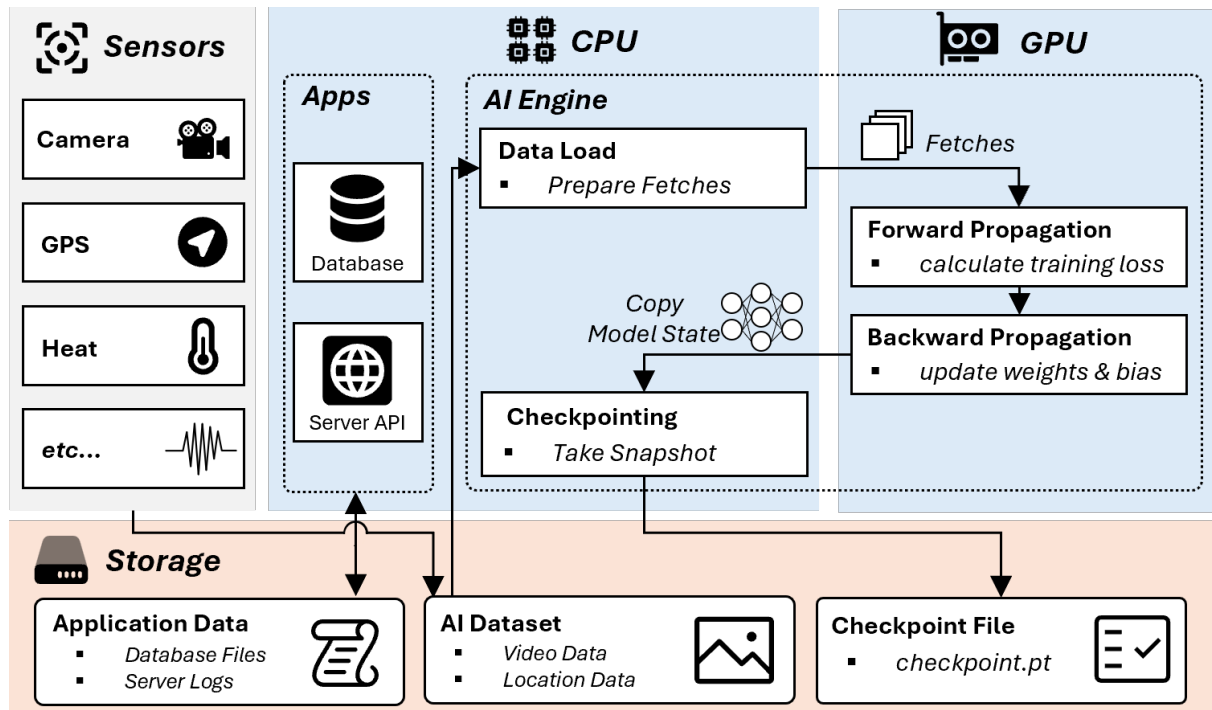
In this section, we briefly explain the training step in the DL framework and then introduce the issues of *checkpoint delay* and waste of storage space that can arise in this step. Finally, we give an overview of related works.

### 2.1. Training and Checkpoint Mechanism in ML model

In general, a deep learning model comprises training and inference steps [29]. An inference step generates predictions based on the information transferred from the training step, which means that the training step is a requirement for making predictions.

Therefore, we briefly explain how the training step works step by step (see Figure 1). The training step can be categorized into three phases: data loading, forward propagation, and backward propagation with weight updates [30]. The first phase is data loading, during which the data stored in the underlying storage media or remote storage devices are read. This phase includes the overheads from the I/O operations needed to read data through storage or network paths. After completing the data loading phase, the intermediate variable values in the neural network model, ranging from the input layer to the output layer, are calculated, resulting in the calculated training loss; we call this

forward propagation. Once the forward propagation phase is complete, the backward propagation phase is triggered to update the network's weights and biases, which helps find the correct answer by computing the gradient of the loss. In the traditional approach, the training step is performed in a serialized manner and is repeated until a pre-defined threshold is satisfied; the threshold is pre-defined by a user and is calculated based on the number of epochs and mini-batches [31,32]. Note that the training step can take a long time to complete, depending on the number of layers and the complexity of the problem to be solved. Unfortunately, this step does not provide persistence of the data because all variable values are stored in the host and GPU memory.



**Figure 1.** Training process of an ML model in multi-tenant cloud environments.

A checkpoint mechanism is adopted to support the persistence of data, storing a set of variable values on the underlying storage media, such as an HDD or SSD [6,33]. In an ML model, the checkpoint operation provides a full snapshot of in-memory values, including the weights, biases, and gradients of each layer. Therefore, the checkpoint in an ML model helps to quickly recover the last version of the intermediate variable values generated during the training of the ML model after a random crash or system failure occurs. Let us consider a simple example where the training step takes at least 7 days, and a system crash occurs 3 days after the start time of the training. In this example, if the checkpoint operation is available, we can recover the last version's data and resume the next process within a few seconds or minutes after the crash. Therefore, the training step can be completed after 4 days. Otherwise, after the crash occurs, the entire training step would need to be restarted, requiring 7 days to complete.

However, the checkpoint mechanism causes two unavoidable problems: *checkpoint delays* and waste of storage space [6,7]. First, the checkpoint mechanism can postpone the training step to wait for the completion of I/O requests from the storage media and a flush operation for data persistence; we call this a *checkpoint delay*. Unfortunately, the *checkpoint delay* is unpredictable and wastes hardware resources in that it leads to GPU resources being idle during the checkpoint operation. This is because current servers in cloud environments cannot guarantee I/O latency since they simultaneously handle sensitive data among multi-tenant applications by sharing their hardware resources. It is important to note that the data stored during checkpoint operations are available only

when system crashes or power failures occur. In other words, the checkpoint mechanism periodically issues a vast volume of data (e.g., at a granularity of MB or GB) to store values for the immediate variables on the storage media. Thus it can waste storage space and reduce the lifetime of storage devices in the case of SSDs.

## 2.2. Related Works

Nowadays, many researchers focus on designing solutions to optimize storage for deep learning workloads as the sizes of models and datasets grow [6,7,24–27,33,34]. Some researchers have proposed checkpoint methods that reduce capacity and overhead during deep learning workloads [7,34]. Check-N-RUN [7] utilizes quantization algorithms and compression mechanisms for large-scale deep learning models. SSDC [34] selectively saves only the differences in important parameters during the training and recovery processes through an adaptive threshold. CheckFreq [6] maintains overall workload performance by adjusting checkpoint frequency and parallel processing, ensuring that checkpoints are performed within user-defined overhead limits. These methods can optimize the overall workload by creating lightweight traditional serialized checkpoints or processing them in parallel during training.

Training for large-scale deep learning models is performed in data centers [17]. Therefore, many researchers have made efforts to optimize storage capacity and data migration in deep learning workloads besides checkpoints [35–37]. Smart-Infinity [35] quantitatively measures the data migration cost during the storage offloading process, using computational SSDs to overcome the GPU memory capacity limitations of large-scale language models while reducing costs through gradient compression. Hassan N. Noura et al. proposed a new scheme that benefits from compression to reduce the capacity of high-resolution image data used for deep learning [36]. The authors of ZeRO-Infinity [37] proposed a CPU and storage offloading framework to address GPU memory limitation issues during the training of large-scale models, achieving near-maximum storage bandwidth through a bandwidth-aware mapping strategy and storage read-write overlapping. One group of researchers focused on the non-negative matrix factorization model and proposed a novel approach that adeptly snapshots hierarchical information by integrating a deep autoencoder and symmetry regularization [38].

Meanwhile, multi-tenant environments still face challenges caused by resource competition [17,39–41]. Some researchers have proposed scheduling to reduce training and inference latency in deep learning caused by resource competition in multi-tenant environments [42,43]. Additionally, UAVs, which are increasingly gaining attention with the advancement of flight technology, typically store vast amounts of data extracted in real time by multiple aircraft in data centers. Deep learning techniques, such as object detection and object tracking, which utilize these data, are also evolving [10–12,14–16,18,19].

Unfortunately, while multi-tenant environments and deep learning are becoming increasingly intertwined with technological advancements, the issue of storage contention caused by deep learning workloads in multi-tenant environments has not been sufficiently addressed. We believe that our efforts contribute to solving this problem.

## 3. Design and Implementation

To address the issues of the checkpoint mechanism in ML models, we carefully revisited the processes of training and checkpoint operations. We identified several processes that could be enhanced for optimization in terms of storage and performance. Based on these observations, we propose two novel design principles to mitigate the overhead caused by ML models.

**Parallel Step:** We first argue whether it is necessary to process all steps in ML in a sequential manner. The checkpoint step in the ML model spends time performing I/O requests, which may stall other steps in multi-tenant environments because running applications share I/O interfaces simultaneously. Therefore, we inject parallelism into the

basic rules of the traditional checkpoint operation to overlap the time for I/O requests with the learning process; we call this the Check-P operation.

**Data Reliability:** We also argue that the data stored by the checkpoint operations are rarely reused. In other words, the stored data are valuable in just two cases: (1) The data are required after a crash during the training step to recover the last parameters of an ML model. (2) The data become accessible to run transfer learning or deploy a model to other systems. In general, data loss may be tolerated in ML models because the data can be recovered using fine-tuning mechanisms [44–46]. Therefore, we design Check-Q and Check-Z operations that reduce the amount of data by adopting quantization and compression mechanisms. Note that the Check-Q and Check-Z operations require extra CPU cycles to run, but these cycles can be hidden because of the benefits of the Check-P operation. As a result, they will not affect the overall performance of ML steps.

Basically, Check-QZP follows the basic rules of the traditional steps in ML models. Algorithm 1 describes how Check-QZP works in terms of the checkpoint operation. First, when the checkpoint operation is triggered, Check-QZP creates a new thread to overlap the checkpoint operation with the training process. In order to make a snapshot, the Check-P operation first copies the model parameters from the memory on the GPU to the host memory. This snapshot is then transferred to the Check-Q operation to perform quantization; we describe this in more detail in the following section. Finally, Check-QZP compresses the data delivered from the Check-Q operation and stores the compressed data on the underlying storage device.

---

#### Algorithm 1 Actions in Check-QZP.

---

**Input:** parameters: model parameters

- 1: **if** *triggered a checkpoint in main script*
- 2: `create_subprocess(Check-P)`

**Function** Check-P():

- 3: `snapshot ← Copy_parameters()`
- 4: `quantized_parameters ← Check-Q(snapshot)`
- 5: `Check-QZ ← Compression(quantized_parameters)`
- 6: `Write_storage(Check-QZ)`
- 7: **return** true

**Function** Check-Q(snapshot):

- 8: **for** parameters **in** snapshot:
- 9:  $x_{\min} \leftarrow \text{Min\_update}(\text{parameters})$
- 10:  $x_{\max} \leftarrow \text{Max\_update}(\text{parameters})$
- 11: *// N means quantization bit-width*
- 12:  $\text{scale} \leftarrow \frac{x_{\max} - x_{\min}}{2^N - 1}$
- 13: **for** parameters **in** snapshot:
- 14: **for** parameter **in** parameters:
- 15:  $x_q \leftarrow \text{Round}\left(\frac{\text{parameter} - x_{\min}}{\text{scale}}\right)$
- 16: *Store  $x_q$  in quantized\_parameters*
- 17: **return** a set of quantized\_parameters

---

### 3.1. Quantization and Compression for Low Storage Costs

Figure 2 shows the overall training process of Check-QZP. For the checkpoint operation, Check-QZP first performs a snapshot that copies in-memory values on the GPU (e.g., weights, biases, and gradients) to the host memory (denoted by ❶). After completing the snapshot operation, it triggers a series of operations to optimize the storage costs for data persistence. In general, quantization is used to store data at lower bit widths by converting floating-point values (e.g., FP32) to integer values without decimal points (e.g., INT8).

Since the arithmetic of floating-point values is more complex than that of integer values, quantization helps save computing costs for the processor, reduce power consumption, and minimize memory usage. Therefore, we applied the quantization approach to the checkpoint operations to benefit the ML model; we call it the *Check-Q* operation in this paper (denoted by ②). This approach can reduce the amount of data issued to the storage media during each checkpoint operation, resulting in saved time and resources for storage I/O operations, and it can help avoid the I/O conflicts on the storage stacks. For quantization, we utilized the following formula:

$$scale = \frac{x_{max} - x_{min}}{2^N - 1} \tag{1}$$

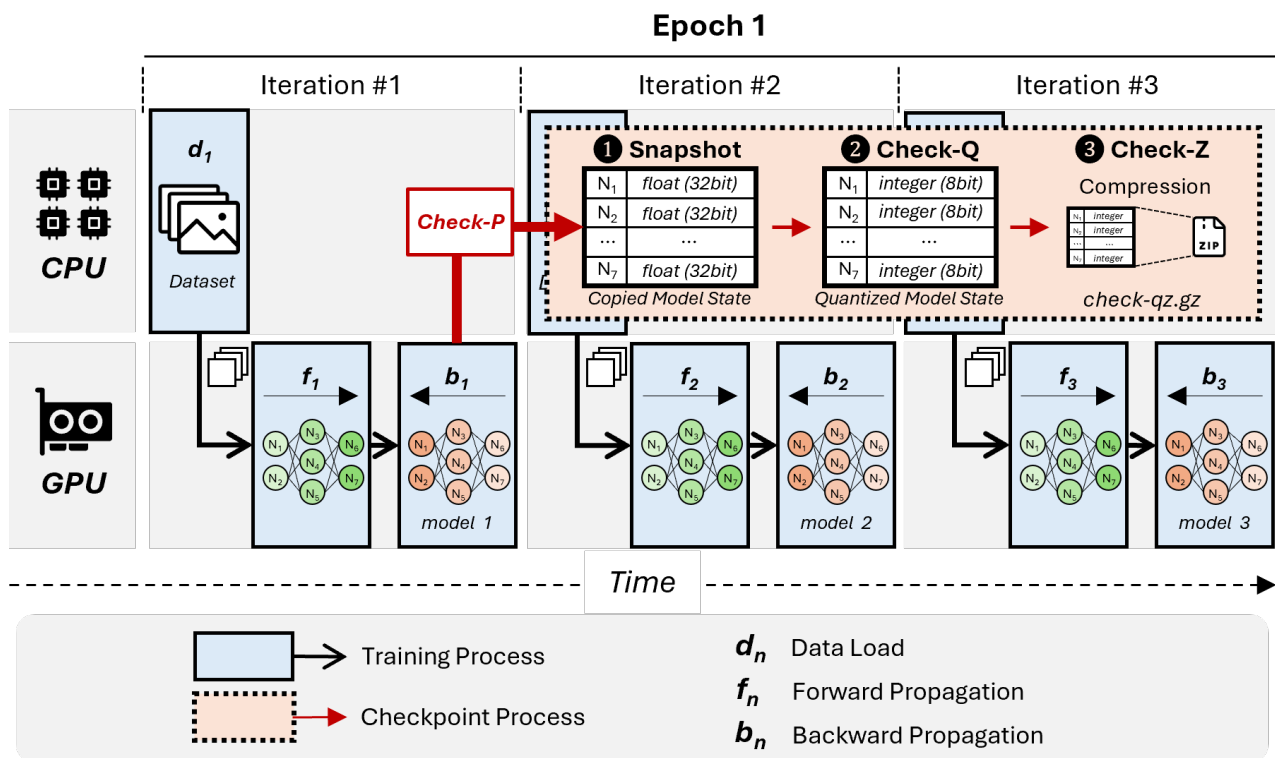


Figure 2. An overview of Check-QZP.

In Equation (1), *scale* calculates the quantization range based on the quantization target  $x$ .  $x_{max}$  and  $x_{min}$  are quantization targets (i.e., maximum and minimum values of model parameters). *scale* is quantized to the size of  $N$  bits.  $x_q$  is a quantized model parameter. It becomes an element normalized to the *scale* range for the quantization target  $x$ . It is expressed in Equation (2):

$$x_q = round\left(\frac{x - x_{min}}{scale}\right) \tag{2}$$

Fortunately, the quantized data offer an opportunity to reduce the amount of data by an order of magnitude through data compression. In other words, after performing *Check-Q* operations, the data are generally encoded as integers in the form of bits without decimal points; thus, there is a high possibility of duplicated neighbors in the quantized data. The compression rate and performance highly rely on the duplication of the data. Therefore, to further reduce the amount of data, *Check-QZP* compresses the quantized data via the *Check-Q* operation once again; we call this operation *Check-Z* in this paper (denoted by ③). To implement the *Check-Q* operation, we utilized the *gzip* [47] library, which is widely used for the compression and decompression of data.

In summary, Check-QZP improves the overall performance of the training step by performing Check-Q and Check-Z operations and prevents the waste of storage space. Of course, these operations have the side effect of CPU costs for performing the above formula and compression, as well as potentially negatively affecting accuracy. Fortunately, any degradation in accuracy can be recovered by using fine-tuning approaches [44–46]. We also believe that the operations of Check-QZP can reduce the cost of I/O operations passing through the storage stack and increase the available storage space. This has a significant impact on I/O performance in multi-tenant cloud environments. Thankfully, the impact of the Check-Q and Check-Z operations is almost zero during normal execution because system crashes and power failures rarely occur in real-world ML scenarios.

### 3.2. Parallelism for High Performance

The Check-Q and Check-Z operations can lead to additional delays in calculating the above formula and compressing the quantized data on the fly. Unfortunately, these operations stall the unintended training step on the GPU because of the serialized order of operations: data loading, forward propagation, backward propagation, and checkpoint operations. To efficiently address this delay, we adopted a parallelism mechanism in Check-QZP that helps to simultaneously run the training process on the GPU, denoted by the blue box with a line, and the checkpoint process on the CPU, denoted by the red box with a dotted line in Figure 2. We call this the *Check-P* operation in this paper. In general, the training process requires a longer duration compared to the checkpoint process; thus, Check-QZP efficiently hides its processing time by overlapping the two processes. Despite this timing behavior, if the training process finishes earlier than the checkpoint operations, we delay running the next training process until the last checkpoint operation is completed; synchronization between the Check-P operation and the training process is ensured using the lock mechanism. In this case, a delay exists to avoid incorrect updating of weight values, but it rarely occurs, as mentioned before.

Finally, for implementing the Check-P operation, we utilized a Python library called *spawn*, which is easy to implement and supports parallelism [48]. Meanwhile, a snapshot for the checkpoint operation must be completed before triggering parallel operations to prevent the partial update of variable values. Therefore, we implemented the Check-P operation to wait until the snapshot operation is complete.

## 4. Evaluation

In this section, we introduce our experimental setup and workloads. Then, we compare Check-QZP with the traditional checkpoint approach to confirm the benefits and limitations of each checkpoint operation. In particular, we aim to answer the following two questions:

- How much can Check-QZP improve the overall performance of the ML model?
- How much storage space does Check-QZP save compared to other methods?

### 4.1. Experimental Setup

We built environments based on a machine (see Table 1) and implemented Check-QZP using the PyTorch framework [22]. Specifically, we implemented the Check-Q, Check-Z, and Check-P operations to isolate the benefits of the compression and parallelism features; some notations were reused together to represent merged operations (e.g., Check-QZP). In the Check-Z operation, we used the *gzip* library to compress or decompress the data generated from the checkpoint operations [47]. We also implemented the multi-object detection (MOD) and multi-object tracking (MOT) models based on Faster-RCNN, which includes the ResNet152 and ResNet50 backbones [49–52].

**Table 1.** Hardware and software environment.

	Description
CPU	Intel® Xeon™ Gold 5215
GPU	NVIDIA Tesla™ T4
Memory	64 GB
Storage	Samsung® SSD 980 PRO (1 TB)
OS	Ubuntu 20.04.6 LTS (64-bit)
PyTorch	version 2.0.0
CUDA	version 11.7

We employed Check-QZP on a UAV dataset, (e.g., VisDrone) [53] with the FIO benchmark [28] to emulate the workload in multi-tenant environments [5,21,41]. Tables 2 and 3 list the detailed information about the dataset.

**Table 2.** Description of dataset.

Dataset	Number of Images	Resolution	Volume
VisDrone Subset1	4561	1904 × 1071	1.14 GB
VisDrone Subset2	1858	1344 × 756	315 MB
VisDrone Subset3	377	2688 × 1512	237 MB

**Table 3.** FIO options.

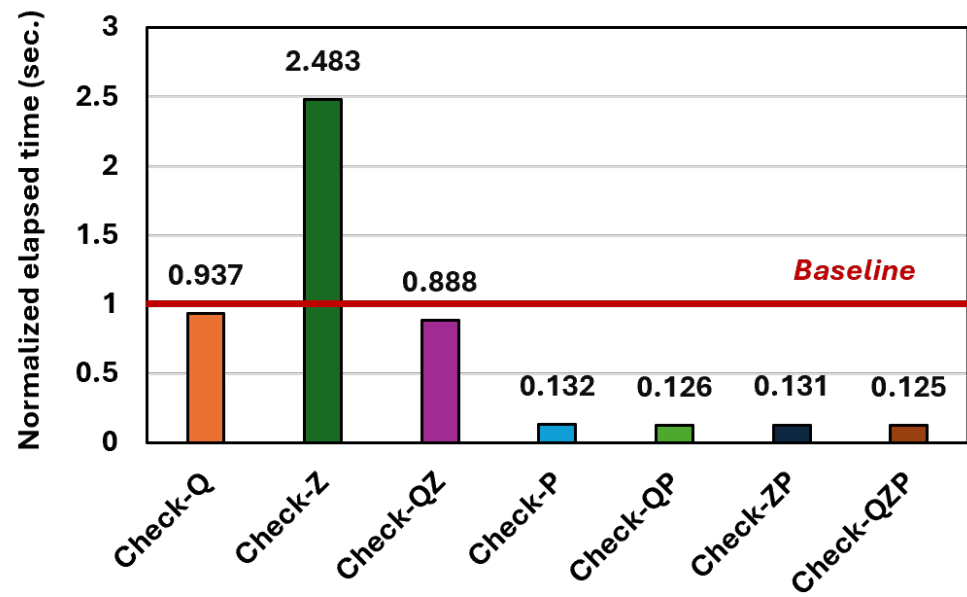
FIO	Options
Ioengine	Libaio
File size	1 GiB
Block size	4 KiB
Direct	False
Numjobs	64
Operation	Random read/write

In this evaluation, we simultaneously performed the FIO benchmark, which includes a 1 GiB file, 4 KiB I/O requests for random read/write, and 64 numjobs. Finally, we carried out experiments by switching checkpoint operations, with each experiment taking more than 14 h to perform model training. Note that the checkpoint operation was triggered in each epoch operation while running the model.

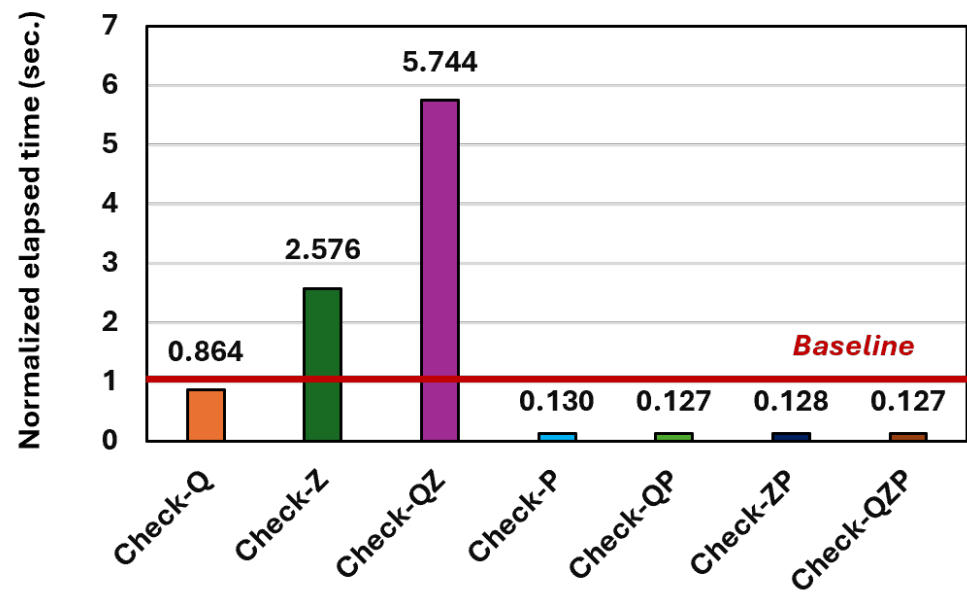
#### 4.2. The Effectiveness of Checkpoint Operations

We first present the effect of Check-QZP on checkpoint operations and describe how it works in a multi-tenant environment. Figure 3 shows the normalized elapsed times of the checkpoint operations. Check-QZP achieved a performance improvement of 87% over the baseline. Check-QZP incorporating parallelism operations exhibited lower delay times compared to compression methods. This is because the delays caused by checkpoint operations can be hidden by overlapping some operations in the training process. In contrast, other methods incorporating compression added delays while waiting for the completion of each checkpoint operation (see Figure 3). In particular, the delay was significantly affected by the I/O intensity. Meanwhile, the compression methods involve a trade-off between the compression rate (i.e., data reduction) and the time required for compression. This trade-off may affect overall performance in multi-tenant environments because multiple applications are performed concurrently and may compete for resources (e.g., CPU and memory). Figure 3 depicts the delays for Check-Z and Check-QZ caused by compression operations.





(a)



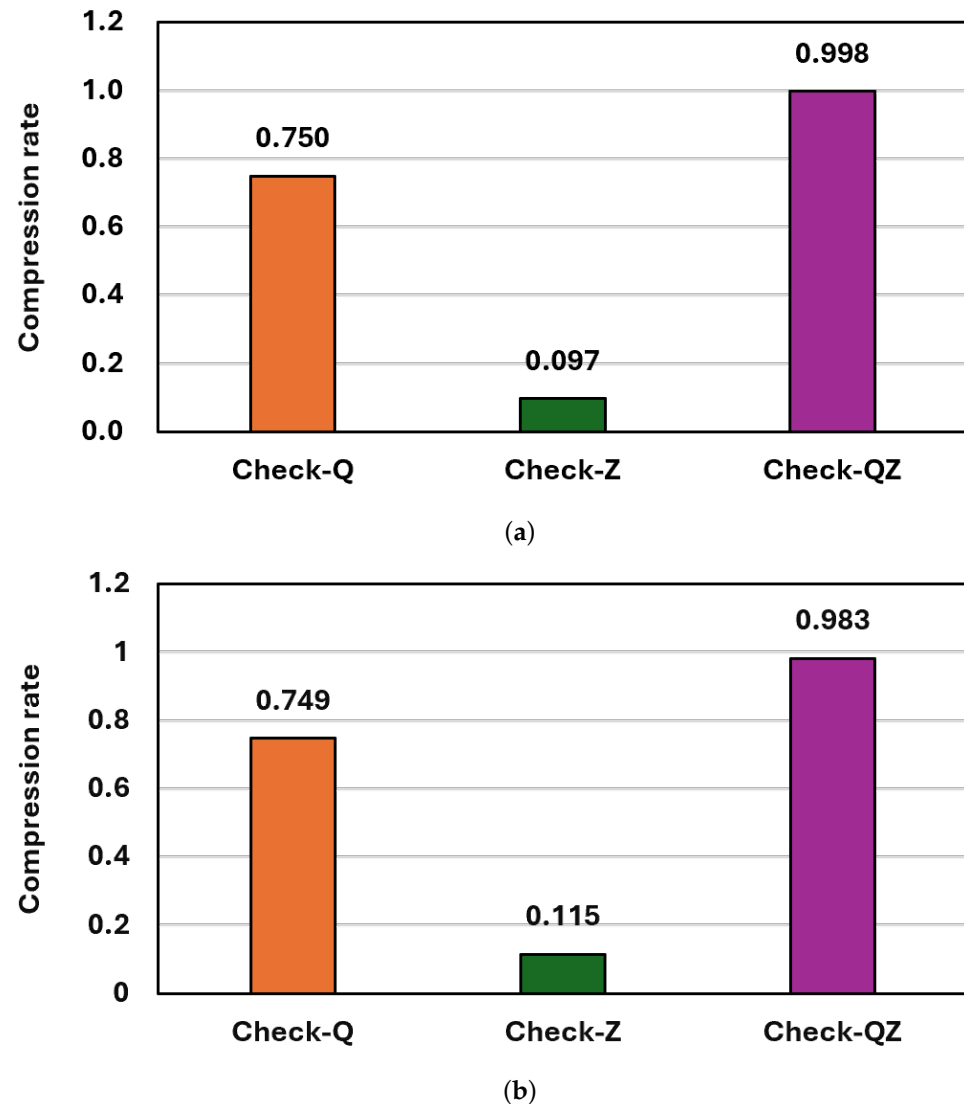
(b)

**Figure 3.** Total checkpoint times for checkpoint methods. (a) Total checkpoint time for MOD model; (b) Total checkpoint time for MOT model.

#### 4.3. Analysis of Compression Rate

Now, we analyze how much storage space is saved when applying compression methods (e.g., compression and quantization) to checkpoint operations. As shown in the previous section, the compression rate is one of the crucial metrics because it determines how much CPU and I/O resources can be saved during checkpoint operations. Figure 4 shows the compression rates of Check-Q, Check-Z, and Check-QZ. As shown in Figure 4, Check-QZ achieved the best compression rate compared to the baseline, Check-Q, and Check-Z; it can reduce the amount of data by up to 99% on average. This outcome can be attributed to the combination of quantization, which represents parameter values with lower bit widths, and gzip, which exhibits the property that the higher the number of repetitive patterns, the greater the compression achieved. Meanwhile, Check-Z, which

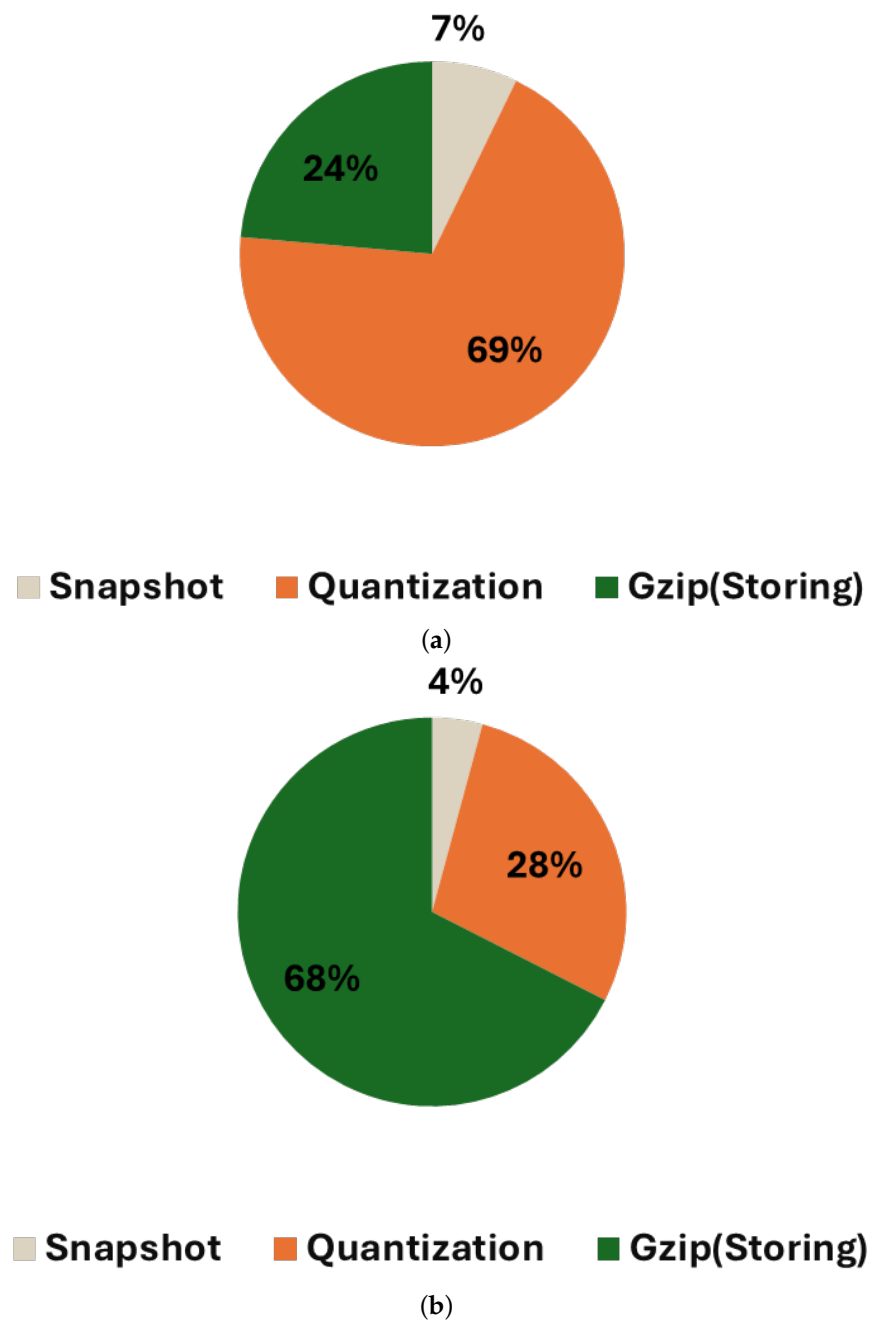
uses only gzip, achieved a low compression rate because it is difficult to find repetitive patterns in FP32. It may incur high overhead due to requiring high CPU utilization to identify and compress the patterns. In contrast, Check-Q and Check-QZ, which utilize quantization technology, demonstrated significant impacts on the compression rate. As shown in Figure 4, Check-Q achieved a compression rate close to 75% by transforming the bit width from 32 bits to 8 bits. It also provides another opportunity to make compression with gzip more effective by reducing the bit range for parameters.



**Figure 4.** Compression rates for quantization and gzip. (a) Compression rate for MOD model; (b) Compression rate for MOT model.

To clearly understand the checkpoint delay, we also measured the composition of the checkpoint delay. Figure 5 shows the composition of the checkpoint delay by snapshot, quantization, and gzip compression. The gzip component includes the compression performed by the gzip algorithm and the storage of compressed data on the underlying storage device. As shown in Figure 5b, gzip compression accounted for the majority of Check-QZ delays. This increase would not compete for bandwidth, since it has no additional write operations beyond the checkpoint operation. The gzip compression delay varied depending on the model structure. As the model became more complex, the checkpoint operation involved more character arrays for mapping parameters. Therefore, the gzip algorithm resulted in an increase in the table size and search time. In contrast, quantization was proportional to the model size since it required only converting time for parameters.

Additionally, as shown in Figure 5a,b, the snapshot component remained at a low rate. As mentioned in Section 3.2, the latency incurred for maintaining consistency in snapshots was expected to have a minimal impact.

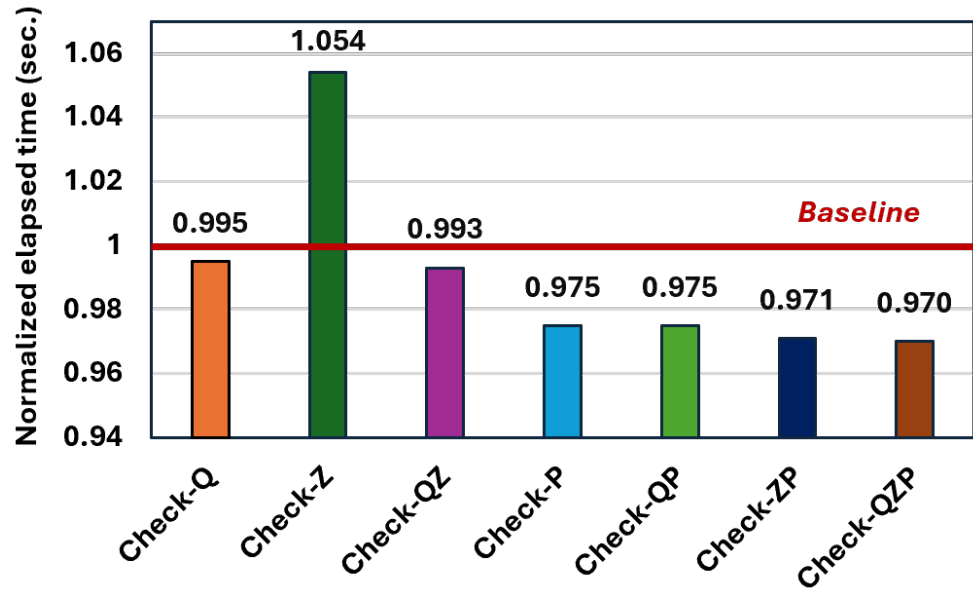


**Figure 5.** Overheads of Check-QZ. (a) Overhead of Check-QZ for MOD model; (b) Overhead of Check-QZ for MOT model.

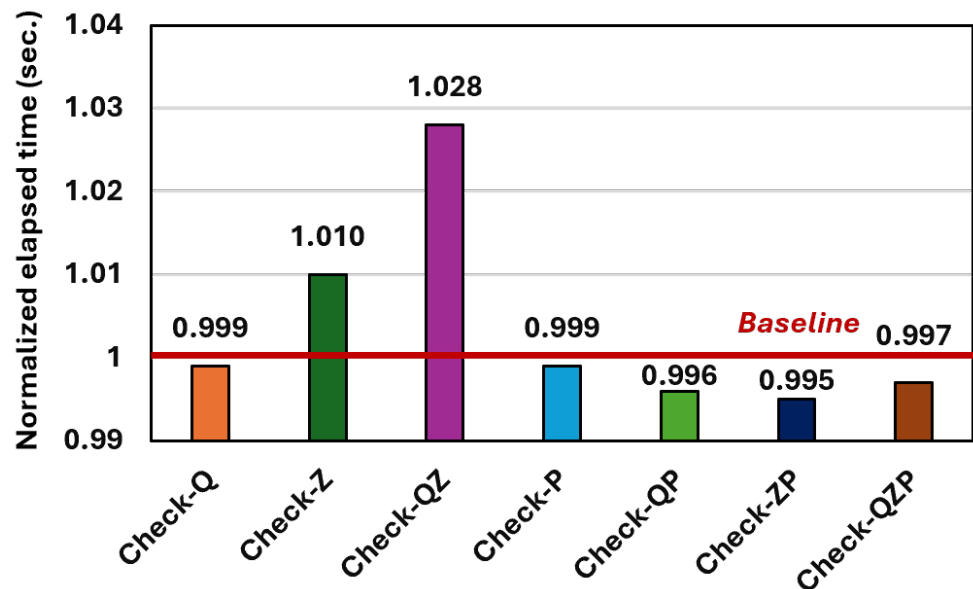
#### 4.4. Overall Training Time

Finally, we measured the overall training time while running the models. Figure 6 shows our evaluation results. As expected, Check-QZP exhibited good performance compared to the baseline. However, the performance gap between the baseline and Check-QZP was significantly reduced compared with the results shown in Figure 3; it only improved the training time by up to 1.5%. These results are reasonable given that the operation time of the checkpoint occupies a very small proportion of the entire training time. This is because the checkpoint operation is triggered after finishing an epoch of the training step, and this

time is very short compared with the training time. In addition, as the training phase runs longer, the proportion decreases in terms of the total training time. In our evaluation, the checkpoint time was responsible for only 1% or 3% of the total training time. We think that these results are a meaningful sign, indicating that Check-QZP can reduce not only overhead but also the amount of data to be stored on the storage device. We also believe that Check-QZP can be applied to a wide range of other models with ease of extension.



(a)



(b)

**Figure 6.** Total training times for checkpoint methods. (a) Total training time for MOD model; (b) Total training time for MOT model.

### 5. Conclusions

As applications increasingly depend on cloud architectures and services, there are negative costs associated with handling excessive I/O operations. These costs are especially harmful to ML-based applications, where a series of training steps can be stalled because of busy I/O scheduling across the storage stacks.

In this paper, we propose a checkpoint mechanism for ML-based applications called Check-QZP, which is designed to provide a strong incentive by adopting three key ideas: quantization, compression, and parallelism. It should be noted that Check-QZP follows the basic rules for the model deployment and maintenance phases of traditional ML processes; thus, it can be easily reused or managed for other models or systems. In addition, Check-QZP can improve the lifetime of NAND-based storage devices by reducing the number of writes to the devices. For our evaluation, we implemented the prototype of Check-QZP using the PyTorch framework, which is commonly used to realize ML technologies. Our evaluation results clearly confirmed that Check-QZP outperformed the baseline in terms of the elapsed time for checkpoint operations. The reason is that Check-QZP can improve the compression rate through quantization and simultaneously utilize resources on both CPUs and GPUs through parallelism.

Finally, we believe that our efforts are valuable and can be applied to various environments based on ML frameworks because checkpoints are necessary for running the training step. In particular, we suggest that Check-QZP can have a positive impact on low-end or heterogeneous systems because they commonly suffer from poor I/O performance [54,55]. Therefore, in the future, we will delve into the performance issues in checkpoint operations and discover additional opportunities for Check-QZP across diverse ML frameworks (i.e., TensorFlow) by analyzing the features of these frameworks. We leave further optimization and detailed studies of Check-QZP's effects on other frameworks for future work.

**Author Contributions:** Conceptualization, S.L. and D.K.; methodology, S.L.; software, S.L.; validation, S.L. and D.K.; formal analysis, S.L. and D.K.; investigation, S.L. and D.K.; resources, S.L., C.L., H.K. and D.K.; data curation, S.L.; writing—original draft preparation, S.L. and D.K.; writing—review and editing, D.A. and D.K.; visualization, S.L., G.M. and D.K.; supervision, D.A. and 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 Gachon University research fund of 2023 (GCU-202400530001) and the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2023-00251730).

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Data sharing is not applicable.

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

## Abbreviations

The following abbreviations are used in this manuscript:

UAV	Unmanned Aerial Vehicle
MOD	Multi-Object Detection
MOT	Multi-Object Tracking
SSD	Solid State Disk
HDD	Hard Disk Drive
AI	Artificial Intelligence
DL	Deep Learning
ML	Machine Learning
DNN	Deep Neural Network
Check-Q	Checkpoint Quantization
Check-Z	Checkpoint Zip
Check-P	Checkpoint Parallelism

## References

1. Pouyanfar, S.; Sadiq, S.; Yan, Y.; Tian, H.; Tao, Y.; Reyes, M.P.; Shyu, M.L.; Chen, S.C.; Iyengar, S.S. A Survey on Deep Learning: Algorithms, Techniques, and Applications. *ACM Comput. Surv.* **2018**, *51*, 92. [CrossRef]
2. Zou, Z.; Chen, K.; Shi, Z.; Guo, Y.; Ye, J. Object Detection in 20 Years: A Survey. *Proc. IEEE* **2023**, *111*, 1–20. [CrossRef]
3. Gwak, M.; Cha, J.; Yoon, H.; Kang, D.; An, D. Lightweight Transformer Model for Mobile Application Classification. *Sensors* **2024**, *24*, 564. [CrossRef] [PubMed]
4. Kwon, C.; Kang, D. Overlay-ML: Unioning Memory and Storage Space for On-Device AI on Mobile Devices. *Appl. Sci.* **2024**, *14*, 3022. [CrossRef]
5. Ebrahim, M.A.; Ebrahim, G.A.; Mohamed, H.K.; Abdellatif, S.O. A Deep Learning Approach for Task Offloading in Multi-UAV Aided Mobile Edge Computing. *IEEE Access* **2022**, *10*, 101716–101731. [CrossRef]
6. Mohan, J.; Phanishayee, A.; Chidambaram, V. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST 21), Online, 23–25 February 2021; pp. 203–216.
7. Eisenman, A.; Matam, K.K.; Ingram, S.; Mudigere, D.; Krishnamoorthi, R.; Nair, K.; Smelyanskiy, M.; Annavaram, M. Check-N-Run: A Checkpointing System for Training Deep Learning Recommendation Models. In Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), Renton, WA, USA, 4–6 April 2022; pp. 929–943.
8. Kennedy, J.; Sharma, V.; Varghese, B.; Reaño, C. Multi-Tier GPU Virtualization for Deep Learning in Cloud-Edge Systems. *IEEE Trans. Parallel Distrib. Syst.* **2023**, *34*, 2107–2123. [CrossRef]
9. Kang, D. Delay-D: Research on the Lifespan and Performance of Storage Devices in Unmanned Aerial Vehicles. *Aerospace* **2024**, *11*, 47. [CrossRef]
10. Hassan, S.A.; Rahim, T.; Shin, S.Y. Real-time UAV Detection based on Deep Learning Network. In Proceedings of the 2019 International Conference on Information and Communication Technology Convergence (ICTC), Jeju, Republic of Korea, 16–18 October 2019; pp. 630–632. [CrossRef]
11. Hu, B.; Wang, J. Deep Learning Based Hand Gesture Recognition and UAV Flight Controls. *Int. J. Autom. Comput.* **2020**, *17*, 17–29. [CrossRef]
12. Bachute, M.R.; Subhedar, J.M. Autonomous Driving Architectures: Insights of Machine Learning and Deep Learning Algorithms. *Mach. Learn. Appl.* **2021**, *6*, 1–25. [CrossRef]
13. Gupta, A.; Anpalagan, A.; Guan, L.; Khwaja, A.S. Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues. *Array* **2021**, *10*, 1–20. [CrossRef]
14. Shin, J.; Piran, M.J.; Song, H.K.; Moon, H. UAV-assisted and deep learning-driven object detection and tracking for autonomous driving. In Proceedings of the 5th International ACM Mobicom Workshop on Drone Assisted Wireless Communications for 5G and Beyond, Sydney, Australia, 17 October 2022; pp. 7–12. [CrossRef]
15. Zhang, S.; Zhuo, L.; Zhang, H.; Li, J. Object Tracking in Unmanned Aerial Vehicle Videos via Multifeature Discrimination and Instance-Aware Attention Network. *Remote Sens.* **2020**, *12*, 2646. [CrossRef]
16. Wu, H.H.; Zhou, Z.; Feng, M.; Yan, Y.; Xu, H.; Qian, L. Real-Time Single Object Detection on The UAV. In Proceedings of the 2019 International Conference on Unmanned Aircraft Systems (ICUAS), Atlanta, GA, USA, 11–14 June 2019; pp. 1013–1022. [CrossRef]
17. Masanet, E.; Shehabi, A.; Lei, N.; Smith, S.; Koomey, J. Recalibrating global data center energy-use estimates. *Science* **2020**, *367*, 984–986. [CrossRef]
18. Liu, J.; Tong, P.; Wang, X.; Bai, B.; Dai, H. UAV-Aided Data Collection for Information Freshness in Wireless Sensor Networks. *IEEE Trans. Wirel. Commun.* **2021**, *20*, 2368–2382. [CrossRef]
19. Gong, J.; Chang, T.H.; Shen, C.; Chen, X. Flight Time Minimization of UAV for Data Collection Over Wireless Sensor Networks. *IEEE J. Sel. Areas Commun.* **2018**, *36*, 1942–1954. [CrossRef]
20. Jeon, M.; Venkataraman, S.; Qian, J.; Phanishayee, A.; Xiao, W.; Yang, F. Multi-tenant GPU Clusters for Deep Learning Workloads: Analysis and Implications. Technical Report, Microsoft Research, 2018. pp. 1–14. Available online: [https://www.microsoft.com/en-us/research/uploads/prod/2018/05/gpu\\_sched\\_tr.pdf](https://www.microsoft.com/en-us/research/uploads/prod/2018/05/gpu_sched_tr.pdf) (accessed on 13 September 2024).
21. Zobaed, S.; Mokhtari, A.; Champati, J.P.; Kourouma, M.; Salehi, M.A. Edge-multiAI: Multi-tenancy of latency-sensitive deep learning applications on edge. In Proceedings of the 2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC), Vancouver, WA, USA, 6–9 December 2022; pp. 11–20.
22. PyTorch. Available online: <https://pytorch.org/> (accessed on 20 July 2021).
23. Tensorflow. Available online: <https://www.tensorflow.org/?hl=en> (accessed on 20 July 2021).
24. Chien, S.D.; Markidis, S.; Sishtla, C.; Santos, L.; Herman, P.; Narasimhamurthy, S.; Laure, E. Characterizing Deep-Learning I/O Workloads in TensorFlow. In Proceedings of the 2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS), Dallas, TX, USA, 12 November 2018; pp. 54–63. [CrossRef]
25. Park, S.; Bahn, H. Performance Analysis of Container Effect in Deep Learning Workloads and Implications. *Appl. Sci.* **2023**, *13*, 11654. [CrossRef]
26. Dey, T.; Sato, K.; Nicolae, B.; Guo, J.; Domke, J.; Yu, W.; Cappello, F.; Mohror, K. Optimizing Asynchronous Multi-Level Checkpoint/Restart Configurations with Machine Learning. In Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), New Orleans, LA, USA, 18–22 May 2020; pp. 1036–1043. [CrossRef]

27. Nicolae, B.; Li, J.; Wozniak, J.M.; Bosilca, G.; Dorier, M.; Cappello, F. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In Proceedings of the 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), Melbourne, Australia, 11–14 May 2020; pp. 172–181. [CrossRef]
28. Axboe, J. Flexible I/O Tester (FIO). 2024. Available online: <https://github.com/axboe/fio> (accessed on 13 September 2024).
29. Choi, J.; Kang, D. Overlapped Data Processing Scheme for Accelerating Training and Validation in Machine Learning. *IEEE Access* **2022**, *10*, 72015–72023. [CrossRef]
30. LeCun, Y.; Bengio, Y.; Hinton, G. Deep learning. *Nature* **2015**, *521*, 436–444. [CrossRef]
31. Goh, A. Back-propagation neural networks for modeling complex systems. *Artif. Intell. Eng.* **1995**, *9*, 143–151. [CrossRef]
32. Li, M.; Zhang, T.; Chen, Y.; Smola, A.J. Efficient mini-batch training for stochastic optimization. In Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY, USA, 24–27 August 2014; pp. 661–670. [CrossRef]
33. Gupta, T.; Krishnan, S.; Kumar, R.; Vijeev, A.; Gulavani, B.; Kwatra, N.; Ramjee, R.; Sivathanu, M. Just-In-Time Checkpointing: Low Cost Error Recovery from Deep Learning Training Failures. In Proceedings of the Nineteenth European Conference on Computer Systems, Athens, Greece, 22–25 April 2024; pp. 1110–1125. [CrossRef]
34. Xiang, L.; Lu, X.; Zhang, R.; Hu, Z. SSDC: A Scalable Sparse Differential Checkpoint for Large-scale Deep Recommendation Models. In Proceedings of the 2024 IEEE International Symposium on Circuits and Systems (ISCAS), Singapore, 19–22 May 2024; pp. 1–5. [CrossRef]
35. Jang, H.; Song, J.; Jung, J.; Park, J.; Kim, Y.; Lee, J. Smart-Infinity: Fast Large Language Model Training using Near-Storage Processing on a Real System. In Proceedings of the 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Edinburgh, UK, 2–6 March 2024; pp. 345–360. [CrossRef]
36. Noura, H.N.; Azar, J.; Salman, O.; Couturier, R.; Mazouzi, K. A deep learning scheme for efficient multimedia IoT data compression. *Ad Hoc Netw.* **2023**, *138*, 102998. [CrossRef]
37. Rajbhandari, S.; Ruwase, O.; Rasley, J.; Smith, S.; He, Y. ZeRO-infinity: Breaking the GPU memory wall for extreme scale deep learning. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, MO, USA, 14–19 November 2021; pp. 1–14. [CrossRef]
38. Zhang, W.; Yu, S.; Wang, L.; Guo, W.; Leung, M.F. Constrained Symmetric Non-Negative Matrix Factorization with Deep Autoencoders for Community Detection. *Mathematics* **2024**, *12*, 1554. [CrossRef]
39. Lv, C.; Yang, L.; Zhang, X.; Li, X.; Wang, P.; Du, Z. Unmanned Aerial Vehicle-Based Compressed Data Acquisition for Environmental Monitoring in WSNs. *Sensors* **2023**, *23*, 8546. [CrossRef] [PubMed]
40. Ebrahimi, D.; Sharafeddine, S.; Ho, P.H.; Assi, C. UAV-aided projection-based compressive data gathering in wireless sensor networks. *IEEE Internet Things J.* **2018**, *6*, 1893–1905. [CrossRef]
41. Ebrahimi, D.; Sharafeddine, S.; Ho, P.H.; Assi, C. Data Collection in Wireless Sensor Networks Using UAV and Compressive Data Gathering. In Proceedings of the 2018 IEEE Global Communications Conference (GLOBECOM), Abu Dhabi, United Arab Emirates, 9–13 December 2018; pp. 1–7. [CrossRef]
42. Zheng, W.; Song, Y.; Guo, Z.; Cui, Y.; Gu, S.; Mao, Y.; Cheng, L. Target-based Resource Allocation for Deep Learning Applications in a Multi-tenancy System. In Proceedings of the 2019 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 24–26 September 2019; pp. 1–7. [CrossRef]
43. Nikolaidis, S.; Venieris, S.I.; Venieris, I.S. MultiTASC: A Multi-Tenancy-Aware Scheduler for Cascaded DNN Inference at the Consumer Edge. In Proceedings of the 2023 IEEE Symposium on Computers and Communications (ISCC), Gammarth, Tunisia, 9–12 July 2023; pp. 411–416. [CrossRef]
44. Rasch, M.J.; Mackin, C.; Gallo, M.L.; Chen, A.; Fasoli, A.; Odermatt, F.; Li, N.; Nandakumar, S.R.; Narayanan, P.; Tsai, H.; et al. Hardware-aware training for large-scale and diverse deep learning inference workloads using in-memory computing-based accelerators. *Nat. Commun.* **2023**, *14*, 5282. [CrossRef]
45. Dettmers, T.; Pagnoni, A.; Holtzman, A.; Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms. *Adv. Neural Inf. Process. Syst.* **2024**, *36*, 1–28.
46. Guo, M.; Dong, Z.; Keutzer, K. SANA: Sensitivity-Aware Neural Architecture Adaptation for Uniform Quantization. *Appl. Sci.* **2023**, *13*, 10329. [CrossRef]
47. Gzip. Available online: <https://www.gzip.org/> (accessed on 20 July 2021).
48. Multiprocessing. Available online: <https://docs.python.org/ko/3/library/multiprocessing.html> (accessed on 20 July 2023).
49. Pal, S.K.; Pramanik, A.; Maiti, J.; Mitra, P. Deep learning in multi-object detection and tracking: State of the art. *Appl. Intell.* **2021**, *51*, 6400–6429. [CrossRef]
50. Dai, Y.; Hu, Z.; Zhang, S.; Liu, L. A survey of detection-based video multi-object tracking. *Displays* **2022**, *75*, 1–17. [CrossRef]
51. Ren, S.; He, K.; Girshick, R.; Sun, J. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Trans. Pattern Anal. Mach. Intell.* **2017**, *39*, 1137–1149. [CrossRef]
52. Zeng, F.; Dong, B.; Zhang, Y.; Wang, T.; Zhang, X.; Wei, Y. MOTR: End-to-End Multiple-Object Tracking with TRansformer. In Proceedings of the European Conference on Computer Vision (ECCV), Tel Aviv, Israel, 23–27 October 2022, pp. 659–675.
53. Du, D.; Zhu, P.; Wen, L.; Bian, X.; Lin, H.; Hu, Q.; Peng, T.; Zheng, J.; Wang, X.; Zhang, Y.; et al. VisDrone-DET2019: The Vision Meets Drone Object Detection in Image Challenge Results. In Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops, Seoul, Republic of Korea, 27–28 October 2019; pp. 1–14.

54. Mokhtari, A.; Hossen, M.A.; Jamshidi, P.; Salehi, M.A. Felare: Fair scheduling of machine learning tasks on heterogeneous edge systems. In Proceedings of the 2022 IEEE 15th International Conference on Cloud Computing (CLOUD), Barcelona, Spain, 10–16 July 2022; pp. 459–468.
55. Filho, C.P.; Marques Jr, E.; Chang, V.; Dos Santos, L.; Bernardini, F.; Pires, P.F.; Ochi, L.; Delicato, F.C. A systematic literature review on distributed machine learning in edge computing. *Sensors* **2022**, *22*, 2665. [[CrossRef](#)]

**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.