CheckFreq: Frequent, Fine-Grained DNN Checkpointing

Jayashree Mohan [*]
UT Austin

Amar Phanishayee
Microsoft Research

Vijay Chidambaram
UT Austin and VMware research

Abstract

Training Deep Neural Networks (DNNs) is a resource-hungry and time-consuming task. During training, the model performs computation at the GPU to learn weights, repeatedly, over several epochs. The learned weights reside in GPU memory, and are occasionally checkpointed (written to persistent storage) for fault-tolerance. Traditionally, model parameters are checkpointed at epoch boundaries; for modern deep networks, an epoch runs for several hours. An interruption to the training job due to preemption, node failure, or process failure, therefore results in the loss of several hours worth of GPU work on recovery.

We present CheckFreq, an automatic, fine-grained checkpointing framework that (1) algorithmically determines the checkpointing frequency at the granularity of iterations using *systematic online profiling*, (2) dynamically tunes checkpointing frequency at runtime to bound the checkpointing overhead using *adaptive rate tuning*, (3) maintains the training data invariant of using each item in the dataset exactly once per epoch by checkpointing data loader state using a *light-weight resumable iterator*, and (4) carefully pipelines checkpointing with computation to reduce the checkpoint cost by introducing *two-phase checkpointing*. Our experiments on a variety of models, storage backends, and GPU generations show that CheckFreq can reduce the recovery time from hours to seconds while bounding the runtime overhead within 3.5%.

1 Introduction

Deep Neural Networks (DNNs) are widely used in many AI applications including image classification [20, 23, 41], language translation [46], and speech recognition [17]. While DNNs have facilitated state-of-the-art accuracy in these tasks, they come at the cost of high computational complexity, taking up to several days to train [8, 35].

Training starts with a randomly chosen set of learnable parameters (such as weights and biases) and proceeds in iterations consisting forward and backward pass over a minibatch of data. At the end of each backward pass, the learnable parameters are recomputed using the gradients obtained, and updated *in GPU memory*. Training is performed for several epochs, where one epoch is a complete pass over the dataset. At the end of training, the learned parameters are saved to persistent storage for inference.

Due to the large runtime of DNN training, the model weights and optimizer state (collectively, model state) are occasionally written to persistent storage, for fault tolerance; else, an interruption to the job due to process failure, or node crash can wipe out all the job state, resulting in loss of several hours of GPU work. This is termed *checkpointing*. Traditionally, models are checkpointed at epoch boundaries [30].

Interruptions to DNN training jobs are common. Be it dedicated enterprise clusters or cloud instances, failures due to software and hardware errors are inevitable. Prior work has shown that infrastructure and process failures are common in large-scale big data clusters, with a mean time between failure (MTBF) of 4 – 22 hours [19, 27]. Similarly, for GPU clusters, recent study of large-scale DNN training clusters at Microsoft [22] highlight that DNN training jobs encounter interruptions due to infrastructure failure, node crashes, software bugs, and user errors. Over the span of the analysis period (2 months), the mean time between job failures was 45 minutes on average (excluding early failures) in the Microsoft cluster.

Furthermore, a recent trend with cloud providers is the emergence of cost-effective preemptible VMs which are priced 6-8× cheaper than dedicated VMs [9, 16, 29]; such VMs may be preempted at any time. Recent work shows that GPU VMs may be preempted as frequent as every 15 minutes and atleast every 24 hours on the Google Cloud [31].

When interruptions occur, the long running, stateful, DNN job terminates abruptly, wiping out the model parameters in-memory. For instance, training ResNext101 to accuracy on ImageNet-1K dataset using a V100 GPU takes 270 hours (~3.9 hours per epoch) [35]; if checkpointing is performed at epoch boundaries, about two hours of GPU computation is wasted on average for every interruption. More generally, there is a trend of growing size of datasets [3, 7, 24], and larger, complex model architectures [8, 10, 35], consequently increasing DNN epoch time and overall training time. Therefore, it is critical to frequently checkpoint training progress, at a finer granularity than epochs *i.e.,* at iteration level. In this work, we explore how to perform fine-grained checkpointing automatically in a model- and hardware-agnostic manner, without intrusive changes to the training workload.

We present CheckFreq, a fine-grained checkpointing framework for DNN training. CheckFreq strikes a balance between ensuring a low runtime overhead and providing a high checkpointing frequency, so that there is minimal loss of GPU time in the event of job interruptions or failures by performing iteration-level checkpointing. CheckFreq has two major components; a checkpointing policy that automatically deter-

[*]Work done as part of MSR internship

mines *when to checkpoint*, and a checkpointing mechanism that performs *correct, low-cost checkpointing*. To this end, we build upon a set of techniques from the High Performance Computing (HPC) and storage community, alongside novel DNN-specific optimizations such as pipelined in-memory snapshots, utilizing spare GPU capabilities for fast snapshot, and a DNN-aware systematic profiling for dynamic tuning of checkpointing frequency. Using CheckFreq, we show that the recovery time reduces from hours to seconds during job interruptions.

Fine-grained checkpointing for DNNs at iteration granularity poses several unique challenges which CheckFreq addresses as described below.

1. **Checkpointing frequency**. There is no single checkpointing frequency that works across models, hardware, and training environments. The frequency of checkpointing depends on several factors; *e.g.,* model size, storage bandwidth, and training iteration time. Moreover, a job could face interference while writing checkpoints due to reading the dataset from the same storage device, or due to concurrently running jobs that share the storage bandwidth to write checkpoints. Statically determining a checkpointing frequency is sub-optimal for runtime if a job faces interference in its training environment.

Therefore, CheckFreq algorithmically determines an initial checkpointing frequency by profiling the job characteristics during runtime. CheckFreq uses *systematic online profiling* to determine the best-case checkpointing frequency for the model in the given training environment. However, in practice, the job might incur additional overheads due to intereference which slows down the checkpointing process. To tackle this, CheckFreq introduces *adaptive rate tuning* to dynamically monitor the job runtime between checkpoint intervals, and appropriately scale up or scale down the checkpointing frequency, so that the end-to-end runtime overhead is within a user-given bound.

2. **Checkpoint stalls**. The model state to be checkpointed is updated every iteration. Therefore, training has to briefly pause to accurately checkpoint the current state; the GPU (or any accelerator) remains idle until checkpoint completes, introducing *checkpoint stalls* in training. Naively increasing the frequency of checkpointing (*e.g.,* every iteration) results in high runtime overhead due to checkpoint stalls.

CheckFreq reduces checkpoint stalls using a *DNN-aware two-phase checkpointing* strategy. The checkpointing operation is split into a `snapshot()` and a `persist()` phase. In the `snapshot()` phase, CheckFreq performs a consistent in-memory copy of all the learnable model state. This operation is *pipelined* with compute until the weight update of the subsequent iteration which is the latest point when the model parameters are updated. In the `persist()` phase, the snapshot is asynchronously written to the storage device. CheckFreq guarantees that a checkpoint is reliably persisted on disk (using `fsync()`) before the subsequent checkpoint operation begins. Therefore, in the event of an unexpected interruption, the job state will rollback at most one checkpoint.

3. **Data invariant**. For a large class of models that perform random data pre-processing operations in every epoch of training (eg CNNs), it is crucial to ensure the following data invariant holds: every epoch must process *all* the items in the dataset *exactly once*, in a random order, with random pre-processing like crop, resize etc. Existing data iterators in frameworks like PyTorch, and MxNet do not support resumability. When the job is interrupted, these iterators can either miss out, or repeat data items in an epoch, resulting in loss in model accuracy when resuming at iteration granularity.

To address this challenge, CheckFreq introduces a resumable data iterator that respects the data invariant even in the presence of interruptions. The iterator uses epoch seeded psuedo-random transformations, that can reconstruct the iterator state as it was prior to interruption. CheckFreq's iterator thus makes correct, iteration-level checkpointing feasible.

We implement CheckFreq as a pluggable module for PyTorch, with minimal (< 10 LOC) changes to the original job's script. Our evaluation across a variety of models, GPUs, and storage types confirms that CheckFreq reduces the wasted GPU time from order of hours to just under a minute, while incurring less than 3.5% runtime overhead, as compared to the existing epoch-based checkpointing schemes. CheckFreq reduces the end-to-end training time by 2× when training a ResNet50 job on a 1080Ti GPU, and by 1.6× for a ResNext101 job on a V100 GPU, when the job is interrupted every 5 hours in both cases. We further demonstrate the importance of CheckFreq's recoverable iterator by training ResNet18 to accuracy using ImageNet dataset with frequent interruptions (once every 2 epochs) and iteration-level checkpointing; Existing state-of-the-art data loaders like DALI [4] result in up to 13% drop in accuracy while CheckFreq is able to train the model to target accuracy.

In summary, this paper makes the following contributions.

- Analyzes the state of DNN checkpointing today and highlights the need for fine-grained checkpointing and the challenges involved in achieveing it (§3)
- The design and implementation of CheckFreq, an automatic, fine-grained checkpointing framework for DNN training that exploits the DNN computational model to provide low-cost, pipelined checkpointing (§4)
- Experimental results demonstrating the efficacy of CheckFreq in reducing the recovery time from hours to seconds, across a range of models and hardware configurations (§5)

2 Background

This section provides a brief overview of the DNN computational model and the role of checkpointing in DNN training.

DNN computational model. Training a Deep Neural Network (DNN) is the process of determining the set of weights and bias in the network, collectively called the *learnable pa-*

rameters. Once trained, the DNN computes the output using the weights learned during the training phase.

DNN Training starts with a randomly chosen set of learnable parameters and proceeds iteratively in steps called *iterations*. Every iteration processes a small disjoint subset of the dataset called a *minibatch*. When the entire dataset is processed exactly once, an *epoch* is said to be complete. Each iteration of training performs the following steps in order.

- **Data augmentation**. Fetches a minibatch of data from storage and applies random pre-processing operations. For *e.g.,* in popular image classification models like the ResNets, pre-processing includes randomly cropping the input image, resizing, rotating, and flipping it.
- **Forward pass**. The model function is applied on the minibatch of data to obtain the prediction.
- **Backward pass**. A loss function is used to determine how much the prediction deviates from the correct answer; each layer in the DNN computes a gradient of the loss.
- **Weight update**. Using the gradients computed in the backward pass, the learnable model parameters are updated.

At the end of training (typically after a fixed number of epochs), the final learned parameters are saved to persistent storage. To perform inference on the model, the DNN is initialized with the learned parameters and the output is predicted.

Checkpointing. Training a DNN is a highly time-consuming task. For instance, BERT-large, the state-of-the-art language modeling network, takes 2.5 days to train [8], when trained in parallel across 16 V100 GPUs. Since the learnable parameters are maintained in GPU memory during training, any interruption to the training job due to a process crash, server crash, job or VM preemption, or job migration, results in the *loss* of model state learned so far. This state is typically a few hundred MBs to a few hundred GBs in size [36] (§5.4). Consequently, several hours of GPU time spent on training will be lost. To overcome this, the model state is typically *checkpointed* at epoch boundaries; *i.e.,* written out to persistent storage for fault-tolerance. This checkpoint can then be loaded when the training job resumes to ensure that progress is not entirely lost.

Recovery Time. When a DNN training job is interrupted, it rolls back to the last completed epoch that was checkpointed as shown in Figure 1. Note that, all the GPU work performed between the last checkpoint and the point of interruption is *lost* and has to be *redone* when training resumes. The amount of GPU time lost due to an interruption is termed the *recovery time*. In other words, this is the time spent to bring the model to the same state as it was prior to the interruption.

3 The Current State of Checkpointing

We analyze the current state of checkpointing in popular open source ML training frameworks like PyTorch [5], TensorFlow [6], and MxNet [11]. We analyze training workloads from MLPerf submissions v0.7, and the official workloads released by NVIDIA, TensorFlow and PyTorch. We find that checkpointing in open source ML training frameworks is incorrect and inefficient.

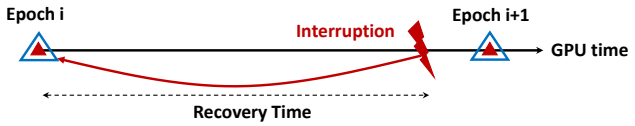

Figure 1: **Recovery time**. The amount of GPU work lost and has to be *redone* on recovery is termed the recovery time.

- **Correctness**. The checkpointing mechanism used in the training scripts could result in loss or corruption of checkpoint files in the event of job failure or interruption.
- **Efficiency**. Checkpointing is inefficient. The frequency of checkpointing is determined in an ad-hoc fashion, typically at epoch-boundaries which results in loss of several hours of GPU time for recovery. Furthermore, there is lack of support for checkpointing at fine granularity; existing data iterators do not support resuming training state at iteration boundaries and results in high checkpoint stalls.

3.1 Checkpointing is Incorrect

Corruption due to overwrites. Some of the official training workloads maintained by PyTorch [38], overwrite the same checkpoint file at the end of each epoch to reduce storage utilization. However, this exposes the risk of corrupting the checkpoint file in the event of a crash during the checkpoint operation. Prior work [37] has shown that different filesystems treat overwrites differently; a crash could result in non-atomic data update in the writeback mode of ext3 resulting in data corruption, while it could truncate the file on ext4, resulting in data loss. In either case, the checkpoint file becomes unusable; training has to restart from the first epoch.

The checkpoint file may not persist. Analyzing the primitives used by training frameworks for checkpointing, such as `torch.save` reveal that they do not `fsync()` the checkpoint file. We verified that this can lead to data loss. Moreover, naively performing frequent synchronous `fsync()` affects training performance significantly (§5.3.1).

3.2 Checkpointing is Inefficient

Checkpointing is performed sparingly in an ad-hoc fashion. There is no systematic checkpointing policy in the training jobs; checkpointing interval is chosen in an ad-hoc fashion. For example, some jobs do not checkpoint during training, while some others start checkpointing only after a large number of epochs (60% of training) have elapsed. In general, we observe that checkpointing is typically performed at epoch boundaries, providing only modest fault-tolerance; in the event of a job interruption, the training will resume from the last completed epoch, which potentially loses several hours of GPU training time that has to be redone. For instance, when ResNext101 is trained using ImageNet on a V100 GPU, *two* hours of GPU time is lost on average if the

job is interrupted (§5.5).

A naive frequent checkpointing schedule results in checkpoint stalls. Providing higher fault-tolerance requires checkpointing to be performed more frequently than at epoch boundaries; *i.e.,* at iteration boundaries. However, naively increasing the frequency of checkpointing introduces a large checkpoint stall in training. Since model weights are constantly updated between iterations, checkpointing requires the training to briefly pause to capture the model weights accurately. We term this overhead (i.e, the time GPU is idle, waiting for the checkpoint to complete) as the *checkpoint stall*. Therefore, it is crucial to find the correct checkpointing frequency given a DNN (because the size of checkpoint varies from 100MBs to 100GBs across DNNs), and the storage bandwidth, to minimize checkpoint stalls.

Violating the data invariant during training can affect model accuracy. Each epoch performs a full pass over the dataset, in a random order and holds the invariant that *each data item is seen exactly once per epoch*. One of the benefits of checkpointing at epoch boundaries is that, the data iterator state need not be persisted, as it is reset at the end of epoch. Checkpointing at a finer granularity (i.e. at iterations), requires infrastructure support to resume the state of data iterator as well. We note that the support to persist iterator state exists in some custom dataloaders of NLP models which do not perform random pre-processing operations for every batch. However, for image and video models that apply random transformations on the input data every batch, the existing dataloaders in PyTorch, MxNet, and state-of-the-art data pipelines like NVIDIA's DALI are *not resumable* at iteration boundaries. As a result, they violate the data invariant in the presence of interruptions, resulting in upto the 13% drop in accuracy for popular models ResNet18 (Fig 6).

3.3 Summary

In summary, we observe that the checkpointing mechanism today is incorrect; resulting in potential checkpoint data loss or corruption. Additionally, the checkpointing policy is ad-hoc; there is no systematic way of determining how frequently one must checkpoint, to both minimize recovery time and incur low checkpoint stalls.

The solution to minimize recovery time is to perform frequent, iteration-level checkpointing. However, performing correct and efficient fine-grained checkpointing is challenging. We need (1) low-cost checkpointing mechanisms, (2) light-weight, resumable data iterators that preserve the model accuracy, and (3) a way to systematically determine the frequency of checkpointing.

4 CheckFreq: Design and Implementation

We present the goals of CheckFreq and the recovery guarantees it provides. We then present an overview of the overall architecture of CheckFreq, and discuss the techniques used by CheckFreq to achieve the enlisted goals.

Technique	*Benefits*
Checkpointing mechanism (How to checkpoint?)	
2-phase checkpointing	Splits checkpointing into two phases and pipelines them carefully with compute to make checkpoints cheap
Recoverable data iterator	Maintains data invariant, allows resuming training at iteration boundaries without affecting accuracy
Checkpointing policy (When to checkpoint?)	
Systematic online profiling	Automatically determines checkpointing frequency, cognizant of model characteristics
Adaptive rate tuning	Dynamically tunes checkpointing frequency to reduce overhead due to interference

Table 1: **Overview of techniques used by CheckFreq**.

4.1 Goals

Correctness. CheckFreq aims to provide frequent, iteration-level checkpointing that is consistent, and persistent.

No impact on model accuracy. CheckFreq aims to not impact the statistical efficiency of the model by ensuring that the data invariant holds when training resumes after interruption.

Automatic frequency selection. CheckFreq aims to determine and tune the frequency of checkpointing *automatically* based on the model being trained, and the training environment (GPU gen, storage type, iteration time). Checkpointing frequency influences the recovery time, *i.e.,* time to bring model state to what it was prior to the interruption.

Low checkpoint stalls. CheckFreq aims to reduce checkpoint stalls during training, so that there is low runtime overhead to frequent checkpointing (*e.g.,* $< 5\%$).

Minimal code changes. CheckFreq aims to require minimal changes to the training code to automate checkpoint management and restoration.

4.2 CheckFreq Recovery Guarantees

An interrupted job resumes training from the latest available checkpoint on disk. In the traditional epoch-based checkpointing, irrespective of when the job is interrupted, training resumes from the previous epoch boundary as shown in Fig 1. If a job performs n iterations per epoch and takes time t_i per iteration, then the average recovery time R_{avg} for this job is :

$$R_{avg} = \frac{n}{2} * t_i$$

This is because, when interrupted in the middle of an epoch, work done so far in the epoch must be redone when resumed, as the state is reset to the end of previous epoch. Thus, recovery time R for epoch-based checkpointing is bounded by:

$$0 \leq R \leq n * t_i$$

Note that $n * t_i$ is the duration of an epoch; it can be as large as a few hours. CheckFreq aims to provide a tight bound

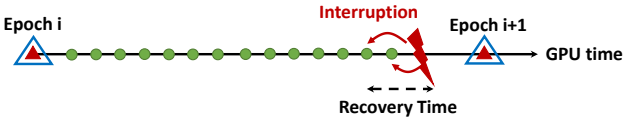

Figure 2: **Bounding recovery time**. CheckFreq guarantees that training rolls back at most one checkpoint.

on recovery time and takes a more fine-grained approach to checkpointing at iteration boundaries. CheckFreq guarantees that *there is at most one ongoing checkpoint operation in the system at any point in time*. When interrupted, it *rolls back at most one checkpoint* - either the last initiated checkpoint (if it completes), or the one prior as shown in Fig 2. If the frequency automatically determined by CheckFreq is k iterations, then CheckFreq guarantees that the recovery time R is bounded by

$$0 \leq R \leq 2 * k * t_i$$
$$R_{avg} = k * t_i \quad (k << n)$$

The chosen checkpointing frequency k is 100 – 300× less than n, as we show later in evaluation (§5.4), thereby resulting in orders of magnitude reduction in recovery time compared to epoch based checkpointing.

4.3 Design

We now present an overview of the architecture of CheckFreq and how it uses various techniques to provide frequent checkpointing at a bounded cost described in §4.2. Table 1 lists the different techniques used by CheckFreq and the benefit of each technique.

Overview. The architecture of CheckFreq is shown in Figure 3. CheckFreq has three major components; a recoverable data iterator that returns a minibatch of data to the training job, a feedback-driven checkpointing policy that determines when to trigger a checkpoint, and a low-cost checkpointing mechanism that is split into a `snapshot()` and a `persist()` phase. CheckFreq monitors the runtime overhead incurred in each checkpoint interval; this is used as feedback to dynamically tune the checkpointing frequency to ensure that the runtime overhead does not exceed a user-given limit p (*e.g.,* 5%). When interrupted, CheckFreq restores the latest available checkpoint and resumes training. We describe each component in detail below.

4.3.1 Checkpointing Mechanism

DNN checkpointing today is performed synchronously; training is paused until the checkpoint operation is complete. However, synchronous checkpointing introduces large *checkpoint stalls*, which results in large runtime overhead if performed frequently. In other words, the cost of a checkpoint (T_c) is high for synchronous checkpointing. For example, consider a policy that checkpoints every three iterations. The model state is written to disk after the weight update phase which updates weights based on the gradients computed in the backward pass. As shown in Figure 4a, the checkpoint cost is incurred in the critical path, resulting in high checkpoint stalls, which

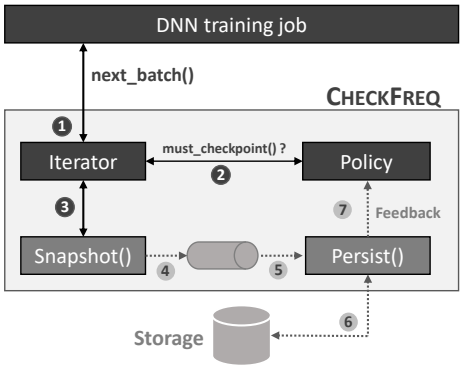

Figure 3: **Training with CheckFreq**. CheckFreq's policy determines the checkpointing frequency. The checkpointing mechanism then snapshots and persists the model and iterator state at the identified frequency in a pipelined manner. If a failure occurs, CheckFreq rolls back the model and iterator state to the latest available checkpoint and resumes training.

can significantly slow down the end-to-end training time. To mask such high checkpoint costs within an overhead p, checkpointing needs to be performed infrequently, which in turn results in high recovery cost.

Two-phase checkpointing. CheckFreq aims to reduce the recovery cost in the event of an interruption by reducing checkpoint stalls. To achieve low checkpoint cost, CheckFreq introduces a *DNN-aware* two-phase checkpointing mechanism. CheckFreq splits checkpointing into two phases; `snapshot()` and `persist()` and pipelines each phase with computation. The main insight behind CheckFreq's two-phase checkpointing is that it exploits the DNN computational model (§2) to pipeline checkpointing operations on modern accelerators such as the GPUs.

1. **Phase 1 : `snapshot()`**. The first is a `snapshot()` phase, performed after the weight update step of the iteration. Here, a copy of the model state is captured in memory, so that it can be written out to storage asynchronously. Since the model state resides in GPU memory, `snapshot()` involves copying the model parameters from GPU to CPU memory. Performing this operation synchronously in the critical path results in non-trivial `snapshot()` overhead as shown in Figure 4b. Therefore, CheckFreq carefully *pipelines* `snapshot()` with compute.

   Pipelining `snapshot()` with compute has to be performed cautiously to ensure consistency of model parameters and preserve correctness of Stochastic Gradient Descent (SGD), which is a popular optimization technique used by learning algorithms. Naively pipelining them can result in an inconsistent snapshot that contains part of the weight updates from one iteration and the rest from the other. CheckFreq exploits the *DNN learning structure* to achieve correct, pipelined snapshots.

   We observe that the learnable model parameters are updated in GPU memory after the backward pass of an

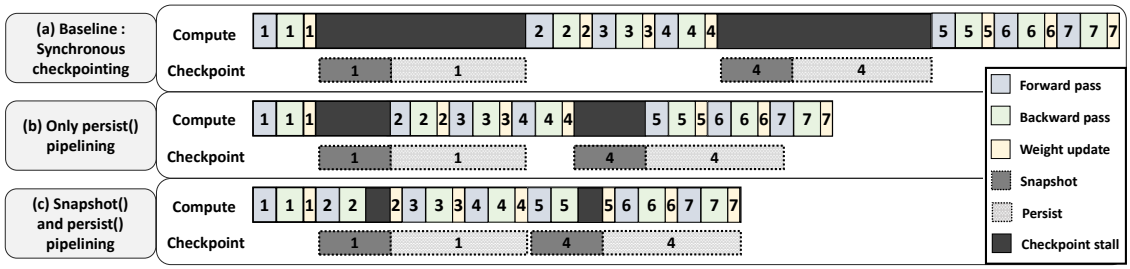

Figure 4: **Pipelining checkpoint with compute**. This figure contrasts three checkpointing mechanisms, when checkpointing is performed every 3 iterations. (a) performs checkpointing synchronously and incurs a high checkpoint stall. (b) takes a snapshot of the model state synchronously but pipelines disk IO (`persist()`) with compute, allowing it to proceed in the background. CheckFreq takes a more nuanced approach by carefully pipelining `snapshot()` with the subsequent iteration's forward and backward pass and incurs lower checkpointing stalls as shown in (c)

iteration; in a step called the *weight update*. Therefore, we can pipeline `snapshot()` of iteration i with compute, until the weight update of iteration $i+1$. If `snapshot()` does not complete by then, then iteration $i+1$ waits until the ongoing `snapshot()` successfully completes as shown in Figure 4c. This tight coupling is required to ensure a consistent snapshot; else we might capture a state that is partially updated by the subsequent iteration that in turn affects the correctness of the learning algorithm [28].

GPU-based `snapshot()`. Although `snapshot()` is pipelined with compute of the following iteration, it may result in checkpointing stalls in cases where it is not possible to completely hide the cost of copying model state from GPU to CPU. Therefore, CheckFreq further optimizes this operation using a GPU-based `snapshot()` when feasible. We observe that the cost of performing a `snapshot()` in GPU memory is an order of magnitude cheaper than performing it to CPU memory, as the latter involves a GPU to CPU copy in the critical path. Therefore CheckFreq takes the following approach.

(a) When spare GPU memory is available in the training environment to hold a copy of the snapshot, we `snapshot()` in the GPU on GPU memory. The `persist()` phase then asynchronously copies the snapshot to CPU memory and then to disk.

(b) If not, CheckFreq snapshots directly into CPU memory. This can introduce stalls in critical path.

(c) CheckFreq adjusts the frequency of checkpointing appropriately to minimize the overhead of `snapshot()`, which can be especially large in (b), and stalls in `persist()`.

2. **Phase 2 : `persist()`**. The second phase in checkpointing is the `persist()` phase which asynchronously writes the snapshot to persistent storage similar to well explored asynchronous checkpointing techniques [33, 34, 40, 45]. However, to provide bounded rollback guarantees discussed in §4.2, `persist()` is *tightly coupled with compute*. CheckFreq performs the `persist()` operation as a background process; and monitors its progress. When a subsequent checkpoint is triggered as determined by the policy, the progress of the ongoing `persist()` operation is checked. If the `persist()` has not completed, then the compute process waits until the ongoing checkpoint operation is complete. This ensures that there is at most one ongoing checkpoint operation at any point in time, and if the job is interrupted, it rolls back to at most one prior checkpoint.

While it may be tempting to abandon an ongoing checkpoint if the next one is triggered, it is a tricky and risky operation. Suppose we abandon the current checkpoint and begin writing the next one, a failure at this point may end up losing both the checkpoints. This could be a chain reaction; a failure could result in rolling back to a significantly old checkpoint if all the recent ones were abandoned, resulting in a high recovery time. Since CheckFreq aims to guarantee that we roll back to at most one prior checkpoint, it does not abandon any running checkpoints.

Resumable light-weight data iterator. The DNN training workload interacts with CheckFreq using a thin API provided by a data iterator. The function of a data iterator in DNN training is to return a pre-processed batch of data items to the GPU, such that the data invariant holds - *each epoch processes all the data items exactly once, in a random order*. While the native iterator in PyTorch and those provided by state-of-the-art data pipelines like DALI [4] support this in the common case, they lack *resumability* if the training is interrupted.

For example, consider a dataset with eight data items from 1 – 8. In an epoch, the order of data items processed could be as shown in Fig 5a. Assume that we checkpoint the model state at the end of every iteration which processes *one* data item. If training is interrupted in the middle of this epoch, the data iterator loses state, and resumes with a random shuffled order of the dataset as shown in Fig 5b, resulting in data items being repeated and missed in a epoch, violating the data invariant.

CheckFreq's data iterator uses the following techniques to support resumption:

- It shuffles data items every epoch using a seed that is a function of the epoch number. Therefore, to recreate the

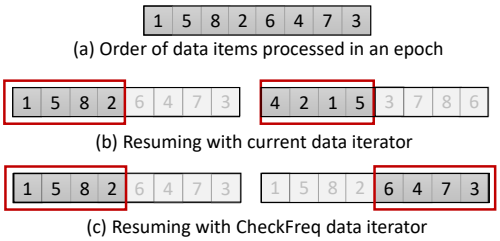

Figure 5: **Resuming iterator state**. When iterator state is not resumable, an epoch might miss data items when job is interrupted (items 3,6,7 are missed in b). CheckFreq (c) ensures that training resumes from exactly where it left off.

same shuffle order, it is sufficient to persist the current epoch ID, and the number of data items processed so far (which makes iterator checkpointing lightweight).

- When training resumes, the iterator reconstructs the shuffle order, and deterministically restarts from where it left off at the last checkpoint as shown in Fig 5c.

Summary. Two-phase checkpointing mechanism along with the resumable data iterator provides correct, low-cost checkpointing. The next important question to answer is, *how frequently should we checkpoint the model?*

4.3.2 Checkpointing Policy

To perform automatic, iteration-level checkpointing, we must determine the frequency at which checkpointing is performed. On one hand, we can checkpoint after every iteration, providing low recovery cost but possibly high runtime overhead. On the other hand, we can perform coarse grained checkpointing at epoch boundaries, resulting in high recovery cost but low runtime overhead. An effective checkpointing policy must find the right balance between recovery cost and runtime overhead, minimizing both. The main idea behind CheckFreq's checkpointing policy is to initiate checkpoints every k iterations (called the checkpointing frequency), such that the overhead of one checkpointing operation can be amortized over k iterations. While prior work in HPC have explored ways of identifying the checkpointing frequency based on failure distribution in the cluster [12, 14, 15], CheckFreq finds the shortest interval that masks the overhead of checkpointing based on the DNN and hardware characteristics.

Systematic online profiling. CheckFreq takes a systematic profile-based approach to determine the checkpointing frequency. It should be chosen such that the runtime overhead introduced due to checkpointing is within a percentage p of the actual compute time, where p is the permissible overhead decided by the user (say 5%).

CheckFreq determines the initial checkpointing frequency as follows. When a training job starts, CheckFreq's data iterator (§4.3.1) automatically profiles several iteration-level and checkpoint-specific metrics which influences the checkpointing frequency - the iteration time (T_i), time to perform weight update (T_w), time to create an in-memory GPU copy (T_g), time

Algorithm 1 : Checkpointing frequency determination

Input: $Ti, Tw, Tc, Tg, Ts, m, M, M_{max}, p$
$T_{oc} \leftarrow max(0, T_c - (T_i - T_w)$
$T_{og} \leftarrow T_g$
if $M_{max} - M > m$ **and** $T_{og} \leq T_{oc}$ **then**
  $T_o \leftarrow T_{og}$
  $mode \leftarrow GPU$
else
  $T_o \leftarrow T_{oc}$
  $mode \leftarrow CPU$
end if
$k \leftarrow \frac{T_c + T_s - T_o}{T_i}$
$k_{min} \leftarrow \left\lceil \frac{T_o}{p * T_i} \right\rceil$
$k \leftarrow max(k, k_{min})$
Output: $k, mode$

to create an in-memory CPU copy (T_c), time to write to storage (T_s), size of checkpoint (m), peak GPU memory utilization (M), and total GPU memory (M_{max}). Based on CheckFreq's 2-phase checkpointing mechanism, the frequency determination algorithm is as shown in Algorithm 1.

The algorithm provides two outputs; 1) the checkpointing frequency k which is the number of iterations elapsed between every checkpoint, and 2) the `snapshot()` *mode* (CPU or GPU-based). The algorithm first determines the snapshot mode based on available free GPU memory; if there is enough space to snapshot the model state in GPU memory, then the mode is set to GPU, else the preferred mode is set to CPU-based snapshotting. Based on the chosen mode, the algorithm estimates the overhead in the critical path incurred after pipelining checkpointing and compute in a tightly coupled manner as described earlier (§4.3.1). It then determines the number of iterations required to amortize this overhead such that the total runtime overhead incurred is below the threshold p. For example, consider the cost of a checkpoint operation and the duration of an iteration are both 1 time unit. If the threshold on runtime overhead is set to 5%, then CheckFreq chooses to checkpoint every 20 iterations.

Adaptive rate tuning. A static, profile-based frequency determination works well when the training environment of the model remains unchanged throughout the runtime of the job. However, in practice, the checkpoint cost estimated by the online profiler can deviate, resulting in higher than estimated runtime overheads. For instance, a job could face write interference by concurrently running jobs sharing storage for read/write, which affects the time to write a checkpoint.

Therefore, CheckFreq uses an adaptive rate tuning technique to perform feedback-driven frequency changes. CheckFreq's iterator monitors the runtime of the job and the actual cost of checkpointing during runtime (after the initial frequency determination). If the observed runtime exceeds the desired overhead, then these values are used to recalculate the checkpointing frequency. The idea is to ensure that the

overall runtime overhead does not exceed the threshold p.

4.4 Implementation

We implement CheckFreq as a pluggable module for PyTorch. The data iterator of CheckFreq is implemented on top of the state-of-the-art data pipeline DALI for PyTorch. CheckFreq can be used as a drop-in replacement to the existing data loader in PyTorch.

CheckFreq determines the initial checkpointing frequency by profiling the first 1% of the iterations in the first epoch, or the first 50 iterations, whichever is the minimum. Therefore, no checkpointing is performed during this initial phase, which is a very small fraction of the total runtime. Additionally, we cache the profiled metrics and the determined policy on persistent storage so that profiling can be skipped when the job resumes after a crash.

CheckFreq internally uses `torch.save()`, followed by a `fsync()` to perform `persist()`, and thus guarantees persistence. To eliminate chances of data corruption, CheckFreq always writes checkpoints to a new file. However, to keep space utilization bounded, CheckFreq only maintains two checkpoints on disk at any given time; one completed checkpoint and the other in-flight. Additionally, checkpoints performed at epoch boundary are preserved (can be turned off by the user). CheckFreq wraps the weight update step in the optimizer with a semaphore that waits on the ongoing `snapshot()` to ensure that a copy of the model state is completed before it is updated by the next iteration.

5 Evaluation

In this section we use a number of microbenchmarks and end-to-end training to accuracy with interruptions to evaluate the efficacy of CheckFreq with respect to the current epoch-based checkpointing scheme across a variety of DNNs. Our evaluation seeks to answer the following questions.

- Can CheckFreq's iterator make iteration-level checkpointing feasible without affecting the accuracy? (§5.2)
- Does CheckFreq's 2-phase checkpoint mechanism reduce checkpoint stalls compared to the existing synchronous strategy? (§5.3)
- Can CheckFreq checkpoint more frequently than epoch-based checkpointing, while incurring low runtime overhead? (§5.4)
- Does CheckFreq reduce the recovery cost when DNN training is interrupted? (§5.5)
- What is the end-to-end benefit of training to accuracy with CheckFreq in the presence of job interruptions in a real preemptive training environment? (§5.6)

5.1 Experimental setup

We evaluate the efficacy of CheckFreq against the state-of-the-art epoch-based checkpointing in PyTorch using the state-of-the-art data pipeline DALI [4].

Servers. We evaluate CheckFreq on two generations of GPU; a Volta V100 GPU with a 1.8TB SSD for persistent storage, and a Pascal 1080Ti GPU with a 1.8TB HDD for persistent storage as shown in Table 2. Both these servers have 8 GPUs, 24 CPU cores and 500GB of DRAM. Both servers run 64-bit Ubuntu 16.04 with CUDA toolkit 10.0 and PyTorch 1.1.0.

	GPU Type	GPU Mem(GB)	CPU Mem(GB)	Storage Media
`Conf-Pascal`	1080Ti	11	500	HDD
`Conf-Volta`	V100	32	500	SSD

Table 2: **Server configurations**. We use two ML server SKUs; each with 24 CPU cores, 500GB DRAM, and 8 GPUs

Models. We use 7 DNNs in our evaluation. ResNet18 [20], ResNet50 [20], ResNext101 [48], DenseNet121 [21], VGG16 [41], InceptionV3 [42] all on Imagenet-1k dataset [39], and Bert-Large pretraining [13] on Wikipedia & BookCorpus dataset [49]. For each model, we use the default minibatch size reported in the literature for these models.

Baseline. We use the epoch-boundary checkpointing as the baseline for all the models except BERT. BERT trains in units of iterations; therefore we use the default checkpointing interval of 200 iterations as the baseline [8]. To perform persistent and correct checkpoints, we explicitly flush the checkpoint file after the checkpoint operation returns.

5.2 Accuracy implications

We first show the need for resumable data iterator to make fine grained iteration-level checkpointing feasible. Using the existing state-of-the-art data iterators to perform iteration-level checkpointing results in violation of the DNN data invariant as described in (§4.3.1). To demonstrate this, we perform the following experiment. We train a ResNet18 job for 70 epochs or to a target accuracy of 69.5% (whichever is earliest) in three different scenarios;

- **No interrupt**. This is the normal training scenario where the job is not interrupted until its completion. There is no checkpointing performed here.
- **Baseline-interrupt**. This scenario uses the existing DALI iterator (same with the native PyTorch iterator) to perform checkpoints at the iteration right before the job is interrupted. We interrupt the job once very 7 minutes (approx every two epochs). This corresponds to commonly used round durations in preemptive schedulers [18, 26, 32, 47].
- **CheckFreq-interrupt**. This setting uses the CheckFreq data iterator that is capable of performing a light-weight checkpoint of iterator state and correctly resuming it. We checkpoint, interrupt, and resume the job exactly as described in the prior setting.

We plot the Top-1 validation accuracy against cumulative training time. Figure 6 shows that it is not possible to perform iteration-level checkpointing using existing iterator, without affecting the model accuracy. This is because, the model state

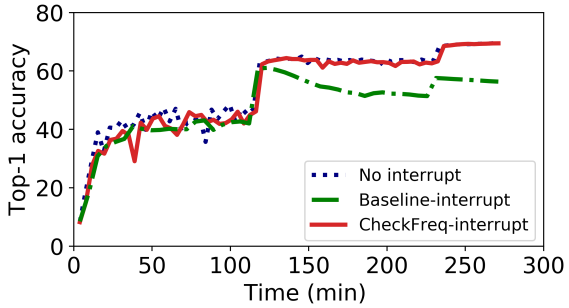

Figure 6: **Impact of resumable data iterator on accuracy**. Performing iteration-level checkpointing with baseline non-resumable data iterator violates the data invariant, results in significant loss of accuracy if job is interrupted. However, CheckFreq's iterator does not affect the final accuracy.

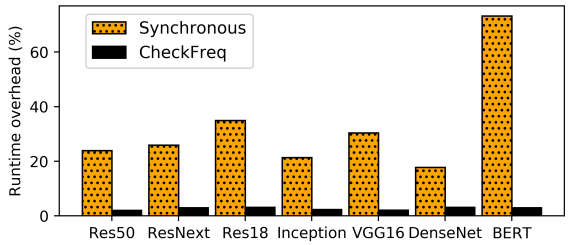

Figure 7: **Runtime overhead for various models**. At a frequency chosen by CheckFreq, synchronous checkpointing incurs upto 70% overhead while CheckFreq's pipelined checkpointing reduces runtime overhead to under 3.5%

is checkpointed at iteration boundaries, but the data loader state is lost. However, with CheckFreq's iterator, the model reaches the target accuracy in the almost the same time as the setting where the job ran without any interruption.

Storage overhead. Checkpointing data iterator state does not have a significant space overhead; it requires persisting two integers - epoch and iteration number, that take up a few bytes on disk. CheckFreq thus provides light-weight, resumable data iterators that do not affect the accuracy of DNNs.

5.3 Performance of checkpointing mechanism

We now evaluate the performance of the two-phase checkpointing strategy of CheckFreq, and compare it against the synchronous strategy. We further provide a split of benefits due to pipelining `persist()` and `snapshot()` operations.

5.3.1 Checkpoint stalls

Figure 7 shows the runtime overhead incurred due to checkpoint stalls with CheckFreq and the baseline checkpointing mechanism while checkpointing at a frequency chosen for that model by CheckFreq on `Conf-Pascal`. The frequency varies across models, but is kept constant for CheckFreq and baseline for a given model. While CheckFreq is able to bound the runtime overheads to about 3.5%, the baseline incurs 17 – 73% runtime overhead due to frequent checkpointing. The reduction in runtime overhead is due to the two-phase checkpointing and pipelining it with computation.

	Checkpoint stall (seconds)		
	Synchronous	***IO pipelining***	***CheckFreq***
`Conf-Volta`	3.6	1.5	0.3
`Conf-Pascal`	10.7	1.3	0.07

Table 3: **Breakdown of benefits**. This table shows the split of checkpoint stall incurred in critical path for VGG16 on two different hardwares

Model	Res18	Res50	ResNext	VGG16	BERT
Freq	147	125	238	83	100
Size(MB)	90	195	482	1055	5000

Table 4: **Checkpoint frequency**. This table shows the number of checkpoints per epoch and the size of each checkpoint

5.3.2 Breakdown of benefits

To understand how much each phase of the checkpointing mechanism contributes to the reduction of checkpoint stalls, we train VGG16 on the two servers using identical batch size of 64 that is the maximum that can fit on `Conf-Pascal`. Checkpointing is performed at a frequency chosen independently for the two servers. We evaluate three settings in Table 3; 1) The baseline synchronous mode, 2) CheckFreq with only `persist()` pipelining (indicated by IO pipelining) and `snapshot()` performed synchronously, 3) CheckFreq with both `persist()` and `snapshot()` pipelining.

On both hardware, CheckFreq is able to significantly reduce the checkpoint cost by 5 – 18× by pipelining both phases of checkpointing with compute as compared to only pipelining `persist()`. On `Conf-Pascal`, the benefit due to pipelining `persist()` is prominent due to the slower storage device. On `Conf-Volta` with fast storage, the CPU cost of `snapshot()` and the storage cost of `persist()` contribute equally to the checkpointing cost. Therefore, pipelining `snapshot()` with compute provides significant speedup.

5.4 Checkpointing policy

We compare the checkpointing frequency determined by CheckFreq for a threshold overhead p of 3.5%. Table 4 shows the number of checkpoints performed per epoch for various models along with per-checkpoint size when performing distributed data parallel training across across 8 GPUs on `Conf-Pascal`. There are two main takeaways here. First, the checkpointing frequency varies with model; therefore frequency selection must take into account the model characteristics. Second, CheckFreq is able to perform 83 – 278× more frequent checkpointing when compared to that performed at epoch boundaries, while incurring $\leq$ 3.5% overhead. On `Conf-Volta`, CheckFreq resulted in 25 – 100× more frequent checkpointing than the epoch-based policy. More frequent checkpoints directly translate to faster recovery times which we evaluate in Section 5.5.

Adaptive tuning of frequency. To demonstrate the importance of adaptive frequency tuning, we perform the follow-

Setting	Isolated	Static	Adaptive
Overhead	5%	**35%**	5%
Frequency (# iterations)	14	14	19

Table 5: **Adpative frequency tuning**. Adaptive frequency tuning is able to dynamically adjust checkpointing frequency to maintain the same overhead as if the job is run in isolation.

Model	*Recovery (seconds)*	
	Baseline	*CF*
ResNet18	840	5
ResNet50	2100	24
VGG16	5700	25
ResNext101	7080	32
DenseNet121	2340	7
Inceptionv3	3000	27
BERT	4920	85

(a) 1 GPU (V100)

Recovery (seconds)	
Baseline	*CF*
180	3
540	8
1320	31
1680	14
600	4
780	42
4500	43

(b) 8 GPU (1080Ti)

Table 6: **Average recovery time (CF - CheckFreq)**.

ing experiment. We run a VGG16 training job on a single GPU (Job-A), allowing it to checkpoint at an initial frequency chosen by CheckFreq (with an overhead of 5%). After 100 iterations have elapsed, we trigger another VGG16 job on a different GPU on the same machine (Job-B), so that the two jobs contend for storage bandwidth to write checkpoints. We measure the runtime for 500 iterations of Job-A with and without adaptive frequency tuning. The results are as shown in Table 5. When Job-A runs in isolation, it incurs an overhead of 5% while checkpointing every 14 iterations. However, when Job-B is introduced after 100 iterations of Job-A, if there is no adaptation across the two jobs, the checkpointing frequency is statically fixed to 14 iterations and the runtime overhead for Job-A increases to 35% (indicated as static in Table 5). This is because, the jobs compete for storage bandwidth, increasing checkpoint cost. In contrast, CheckFreq's adaptive rate tuning dynamically adjusts the checkpointing frequency and keeps the overhead bounded at 5%.

5.5 Recovery time

To understand the benefits of using CheckFreq in the presence of job interruptions, we evaluate the recovery time with the epoch-based checkpointing and CheckFreq. With epoch-based checkpointing, irrespective of when during the epoch the job is interrupted, the job rolls back to the previously completed epoch. Therefore, in the best case, if a failure occurs immediately after the finish of an epoch, then the recovery time is the same as CheckFreq. However, on average, half an epoch's worth of work can be lost if the job is interrupted in the middle of an epoch. And in the worst case, the entire epoch must be redone if the job fails just before the completion of an epoch. For the seven different models, we compare the average case recovery time in two distinct scenarios; 1) a single-GPU training job on `Conf-Volta` in Table 6a and 2) a 8 GPU data-parallel job on `Conf-Pascal` in Table 6b.

As can be seen, CheckFreq is able to reduce recovery time from several minutes (and hours) to just a few seconds, all while incurring less than 3.5% runtime overhead. For instance, when training ResNext101 on a V100 GPU, on average, CheckFreq reduces the recovery time from 2 hours to 32 seconds on average.

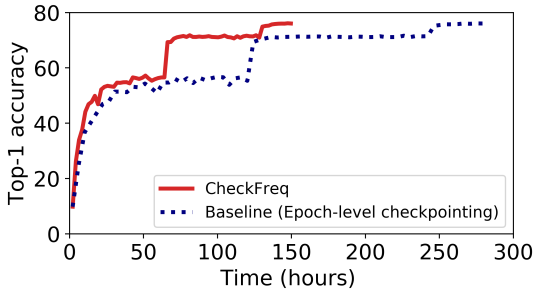

Figure 8: **End-to-end training**. We train Resnet50 using a `Conf-Pascal` GPU with interruptions every 5 hours. CheckFreq trains to state-of-the-art accuracy (76.1%) 2× faster than epoch-based checkpointing by reducing recovery time.

5.6 End-to-end training

We evaluate the end-to-end benefit of training with CheckFreq by simulating a preemptive cluster scenario. We consider a cluster with a premeptive scheduler similar to the one in large production clusters like Philly [2,22]. We consider an average preemption interval of 5 hours. Figure 8 plots the total training duration against top-1 validation accuracy for the epoch-based baseline checkpointing strategy and CheckFreq for training ResNet50 using a GPU on `Conf-Pascal` to state-of-the-art accuracy. CheckFreq results in 2× faster training by reducing recovery time from 1.9 hours to under a minute for every interruption. A similar experiment on `Conf-Volta` resulted in 1.6× faster training time to accuracy for ResNext101.

6 Discussion

Applicability to distributed cluster training. CheckFreq currently works with the distributed data parallel (DDP) mode, where only one GPU per node (rank 0) is responsible for checkpointing. While we show results for single- and multi-GPU training, extending it to multi-node settings is straightforward; checkpointing in multi-GPU and multi-node settings is the same for DDP in frameworks such as PyTorch. Model weights are synchronized across different workers (same node or in the distributed cluster) typically every iteration, or accumulated over a few tens of iterations before synchronizing; therefore each node sees the same version of weights at these synchronization points. Hence, one instance of CheckFreq runs on each node, and persists an identical checkpoint for local recovery at synchronization boundaries. Since each node persists checkpoints independently, and in parallel, there is no additional synchronization overhead for checkpointing.

Generality. CheckFreq focuses on optimizing checkpointing, which is by far the predominant way in which DNN training jobs recover from failures. While our paper focuses on data

parallel training, prior work in model or pipeline parallelism, also rely on checkpointing. Using CheckFreq, checkpointing at minibatch boundaries (every n iterations), each pipeline stage only persists a subset of parameters and optimizer state hosted by that worker. CheckFreq also enables checkpointing within minibatch boundaries during pipeline parallel training (every m microbatches), as CheckFreq's iterator controls the introduction of each microbatch into the pipeline. Checkpointing at the microbatch granularity requires storing additional model state – specifically accumulated weight gradients at every stage in addition to parameter and optimizer state. We leave it to future work to integrate CheckFreq's implementation into frameworks supporting pipeline parallelism.

While we implement CheckFreq in PyTorch, we can extend it to other frameworks like TF and MxNet by wrapping the framework-specific APIs into those exposed by CheckFreq.

7 Related Work

Asynchronous DNN checkpointing. While recent work like DeepFreeze [33] that perform asynchronous DNN checkpointing employ techniques similar to CheckFreq for IO pipelining, it only considers CPU clusters. It does not consider the cost of snapshotting the model state in memory when trained using state-of-the-art GPUs. Our work shows that on modern ML optimized servers, the cost of snapshotting the model state (copying from GPU to CPU) is significant, demonstrating how to pipeline this transfer with compute, and use spare GPU capabilities to enable fast snapshotting.

Furthermore, DeepFreeze requires manual intervention to tune the checkpointing frequency for a given model, hardware and training environment while CheckFreq masks these complexities from the user and analytically identifies the best parameters for checkpointing. Unlike DeepFreeze that uses a static checkpointing frequency, CheckFreq is also beneficial in shared cluster settings, as it adapts the checkpointing frequency based on memory and storage interference due to other jobs to minimize checkpoint stalls.

Asynchronous checkpointing in HPC. Prior work in HPC [34, 40, 45] uses asynchronous checkpointing to mask the IO latency. A key challenge that differentiates DNN checkpointing from traditional HPC ones is that, performing a synchronous in-memory copy of the model state from GPU to CPU is expensive due to the increasingly fast compute capabilities of the GPU. CheckFreq exploits the DNN learning structure to carefully pipeline even the in-memory snapshot with computation to perform correct, consistent checkpointing. Moreover, CheckFreq further reduces the latency of checkpointing by utilizing spare GPU memory and compute capabilities when possible to perform fast snapshots.

Checkpoint interval estimation in HPC. Prior work [12, 14, 15] determine checkpointing interval for large scale HPC applications based on failure distributions observed in the system. CheckFreq does this in a DNN-aware fashion by exploiting the deterministic, repetitive structure of DNN training to systematically profile resource utilization at runtime.

Adaptive checkpointing. The idea of using adaptation for fault management has been used in HPC applications [25] to decide when to checkpoint, based on a failure prediction module. CheckFreq introduces adaptivity in DNN checkpointing frequency. It identifies and dynamically adapts the checkpointing frequency, based on the characteristics of the model being trained, system hardware, and interference due to other jobs.

TensorFlow Checkpoint Manager. TF checkpoint manager [43] allows checkpointing at a user-given time interval, and supports persisting iterator state. However, it has three shortcomings. First, the checkpointing frequency is decided in an ad-hoc fashion by the user; this introduces large checkpoint stalls if not chosen carefully. Second, it cannot checkpoint the iterator state if random data transformation is involved; this is common for most image based models [44]. Finally, even in cases where it can persist iterator state, TF writes the entire operator graph to storage along with prefetched items resulting in large checkpoint size. CheckFreq addresses these challenges by automatically adapting the checkpointing frequency and using a light-weight, resumable data iterator.

Framework-transparent checkpointing. Transparent checkpointing techniques such as CRIU [1] can backup entire VM state for fault-tolerance; however they do not checkpoint GPU or accelerator state. Even if they were to capture entire device state, device state alone is an order of magnitude larger than the model state captured at iteration boundaries, making frequent CRIU checkpoints impractical. Thus, in this work, we focus on the dominant approach to DNN fault-tolerance - framework-assisted checkpointing of model state.

8 Conclusion

This paper presents CheckFreq, an automatic, fine-grained checkpointing framework for DNN training. CheckFreq achieves consistent, low-cost checkpoints at iteration level using a resumable data iterator, a pipelined two-phase checkpointing mechanism, and automatic determination and tuning of checkpointing frequency. When the job is interrupted, CheckFreq reduces recovery time for popular DNNs from hours to seconds, while incurring low runtime overhead.

Acknowledgements

This work was done during an internship at Microsoft Research as part of Project Fiddle. We thank our shepherd Mehul Shah, the anonymous FAST reviewers, members of the UT SaSLab, fellow Project Fiddle interns Youjie Li, Kshiteej Mahajan, Andrew Or, and many of our MSR colleagues for their invaluable feedback that made this work better. We sincerely thank MSR Labs for their generous support in procuring the many resources required for this work. This work was supported by NSF CAREER #1751277 and donations from VMware, Google, and Facebook.

References

[1] CRIU checkpointing. https://criu.org/Main_Page.

[2] Microsoft Philly Traces. https://github.com/msr-fiddle/philly-traces.

[3] Training a Champion: Building Deep Neural Nets for Big Data Analytics. https://www.kdnuggets.com/training-a-champion-building-deep-neural-nets-for-big-data-analytics.html/.

[4] NVIDIA DALI. https://github.com/NVIDIA/DALI, 2018.

[5] PyTorch. https://github.com/pytorch/pytorch, 2019.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016.

[7] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *arXiv preprint arXiv:1609.08675*, 2016.

[8] NVIDIA AI. BERT Meets GPUs. hhttps://medium.com/future-vision/bert-meets-gpus-403d3fbed848.

[9] Amazon. Amazon EC2 spot instances. https://aws.amazon.com/ec2/spot/?cards.sort-by=item.additionalFields.startDateTime&cards.sort-order=asc.

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[12] John T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[14] Sheng Di, Mohamed-Slim Bouguerra, Leonardo Arturo Bautista-Gomez, and Franck Cappello. Optimization of multi-level checkpoint model for large scale HPC applications. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 1181–1190. IEEE Computer Society, 2014.

[15] Sheng Di, Mohamed-Slim Bouguerra, Leonardo Arturo Bautista-Gomez, and Franck Cappello. Optimization of multi-level checkpoint model for large scale HPC applications. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 1181–1190. IEEE Computer Society, 2014.

[16] Google. Preemptible VM instances. https://cloud.google.com/compute/docs/instances/preemptible#preemptible_with_gpu.

[17] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[18] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 485–500. USENIX Association, 2019.

[19] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis,*

SC 2017, Denver, CO, USA, November 12 - 17, 2017, pages 44:1–44:12. ACM, 2017.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[21] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 2261–2269. IEEE Computer Society, 2017.

[22] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[24] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Tom Duerig, et al. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *arXiv preprint arXiv:1811.00982*, 2018.

[25] Zhiling Lan and Yawei Li. Adaptive fault management of parallel applications for high-performance computing. *IEEE Trans. Computers*, 57(12):1647–1660, 2008.

[26] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 289–304. USENIX Association, 2020.

[27] Catello Di Martino, Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 610–621. IEEE Computer Society, 2014.

[28] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. Convergence analysis of distributed stochastic gradient descent with shuffling. *Neurocomputing*, 337:46–57, 2019.

[29] Microsoft. Use low priority VMs. https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms.

[30] MLPerf. MLPerf Training Results v0.7. https://github.com/mlperf/training_results_v0.7.

[31] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Analysis and exploitation of dynamic pricing in the public cloud for ml training. *DISPA*, 2020.

[32] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. *arXiv preprint arXiv:2008.09213*, 2020.

[33] Bogdan Nicolae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*, pages 172–181. IEEE, 2020.

[34] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. Veloc: Towards high performance adaptive asynchronous checkpointing at large scale. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 911–920. IEEE, 2019.

[35] NVIDIA. ResNext101 Training. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Classification/ConvNets/resnext101-32x4d.

[36] OpenAI. GPT-3 Checkpoint. https://github.com/openai/gpt-3/issues/1.

[37] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 433–448. USENIX Association, 2014.

[38] PyTorch. PyTorch Training Examples. https://github.com/pytorch/examples/tree/master/imagenet.

[39] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[40] Faisal Shahzad, Markus Wittmann, Thomas Zeiser, Georg Hager, and Gerhard Wellein. An evaluation of different I/O techniques for checkpoint/restart. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013*, pages 1708–1716. IEEE, 2013.

[41] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[42] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2818–2826. IEEE Computer Society, 2016.

[43] TensorFlow. Tensorflow checkpoint manager. https://www.tensorflow.org/api_docs/python/tf/train/CheckpointManager.

[44] TensorFlow. Tensorflow iterator checkpointing. https://www.tensorflow.org/guide/data#iterator_checkpointing.

[45] Devesh Tiwari, Saurabh Gupta, and Sudharshan S. Vazhkudai. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 25–36. IEEE Computer Society, 2014.

[46] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[47] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018.

[48] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 5987–5995. IEEE Computer Society, 2017.

[49] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.