Optimizing Resource Utilization, Efficiency and Scalability in Deep Learning Systems

by

Xiaofeng Wu

Submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

UNIVERSITY OF TEXAS ARLINGTON

May 2023

Author . . . .
Department of Computer Science and Engineering
April 27, 2023

Certified by . . . .
Jia Rao
Associate Professor
Thesis Supervisor

Accepted by . . . .
Hong Jiang
Wendell H. Nedderman Endowed Professor and Chair

Optimizing Resource Utilization, Efficiency and Scalability in Deep Learning Systems

by

Xiaofeng Wu

Submitted to the Department of Computer Science and Engineering
on April 27, 2023, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

This thesis addresses the challenges of utilization, efficiency, and scalability faced by deep learning systems, which are essential for high-performance training and serving of deep learning models. Deep learning systems play a critical role in developing accurate and complex models for various applications, including image recognition, natural language understanding, and speech recognition. This research focuses on understanding and developing deep learning systems that encompass data preprocessing, resource management, multi-tenancy, and distributed model training.

The thesis proposes several solutions to improve the performance, scalability, and efficiency of deep learning applications. Firstly, we introduce SwitchFlow, a scheduling framework that addresses the limitations of popular deep learning frameworks in supporting GPU sharing and multi-tasking. Secondly, we propose Atom, a distributed training framework for large language models that utilizes decentralized training to reduce communication costs and increase scalability. We discuss the challenges of decentralized training and present the design and implementation of Atom. Lastly, we introduce PerFect, a method that pre-trains the model using repetitive data to improve data processing efficiency and fine-tunes it to achieve the desired accuracy.

Our approach provides a significant improvement in the performance, scalability, and efficiency of deep learning applications. Specifically, SwitchFlow reduces interference and eliminates out-of-memory errors by scheduling subgraphs instead of computation graphs as a whole. Additionally, it allows subgraphs running on different devices to overlap with each other, leading to a more efficient execution pipeline. Atom achieves high training throughput and fault-tolerance in a decentralized environment, enabling the training of massive-scale models using affordable hardware such as consumer-class GPUs and Ethernet. Finally, PerFect improves the throughput performance of the data preprocessing stage and achieves the desired accuracy when reusing cached data, without the need for additional hardware or third-party libraries.

The proposed frameworks and solutions are evaluated using representative DL models, and the results demonstrate their effectiveness and scalability. Overall, this thesis contributes to the development of deep learning systems and provides practical solutions to the challenges of utilization, efficiency, and scalability, making deep learning applications more accessible and efficient for a wider range of users.

Thesis Supervisor: Jia Rao
Title: Associate Professor

Acknowledgments

I would like to express my gratitude to my parents for their unwavering support, encouragement, and love throughout my journey. Their consistent belief in me has been a constant source of inspiration and motivation. Without their invaluable assistance, I could not have accomplished this significant achievement. I am forever grateful for their unwavering support and guidance.

I would like to express my heartfelt gratitude to my supervisor, Dr. Jia Rao, for his support, guidance, and encouragement throughout my Ph.D. journey. His deep insights, constructive criticisms, and valuable feedback have been instrumental in shaping my research work. I am also grateful to him for providing access to state-of-the-art equipment, including GPU servers, which have been crucial in carrying out experiments.

I would also like to extend my sincere thanks to my thesis committee members, Dr. Hong Jiang, Dr. Song Jiang, and Dr. Hao Che, for their invaluable feedback and guidance, which helped me to improve the quality of my research work. Their constructive criticisms and valuable insights have been of great help to me.

I would like to acknowledge the valuable contributions of my collaborators, without whom this research would not have been possible. Their hard work, dedication, and expertise have significantly improved the quality of my work. I am grateful to have had the opportunity to collaborate with such talented individuals, who have challenged me to think beyond my limitations and have broadened my perspective. Their support and encouragement have been a source of motivation for me, and I am truly thankful for their contributions to my research.

Lastly, I want to thank the cities of Dallas, Arlington, and Fort Worth for providing me with a colorful and enriching life. The experiences and memories I gained from living in these cities will always hold a special place in my heart.

Once again, I express my sincere gratitude to all those who have helped and supported me throughout my Ph.D. journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep Learning [65] has emerged as a powerful paradigm in the field of artificial intelligence, enabling the development of highly accurate and complex models for tasks such as image recognition [50], text to image generation [101], natural language understanding [21], speech recognition [96, 122], and structural biology and drug discovery [62]. To effectively utilize deep Learning for these applications, it is crucial to have robust deep Learning systems in place that provide efficient and scalable infrastructure for tasks such as data preprocessing, model training, and resource management. These systems play a critical role in supporting high-performance deep Learning training and serving, and are essential for achieving state-of-the-art results in various domains.

State-of-the-art deep learning systems have made significant advancements in recent years, with various frameworks and tools being developed to address the unique requirements of deep learning training and serving. Examples of such systems include TensorFlow [6, 47, 1, 46, 43], PyTorch [93], and OneFlow [134]. These systems provide efficient and scalable solutions for tasks such as distributed computing, data preprocessing, model training, and resource management, and have been widely adopted by the research and industry communities for developing high-performance deep learning applications.

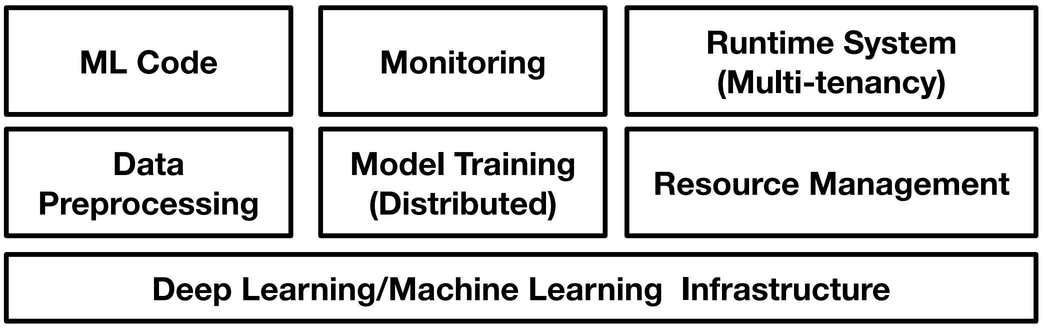

ML Code
Monitoring
Runtime System (Multi-tenancy)
Data Preprocessing
Model Training (Distributed)
Resource Management
Deep Learning/Machine Learning Infrastructure

Figure 1-1: Deep learning system

1.1 Motivation

The increasing demand for high-performance deep learning applications in areas such as computer vision, natural language processing, and speech recognition has led to the development of complex and accurate models that require significant computing resources. Deep learning systems presented in Figure 1-1 have been designed to address these challenges and enable the development and deployment of large-scale deep learning models. However, these systems still face several challenges, including the need for efficient resource management, scalability, and multi-tenancy support.

Efficient resource management is critical in deep learning systems because the large amount of data and computing resources required for training and inference can lead to high costs and long processing times. Scalability is also essential in deep learning systems because models need to be trained on large datasets, and as the amount of data grows, the system must be able to scale to handle the increased workload. Finally, multi-tenancy support is necessary because deep learning systems are often shared by multiple users, each with their own unique requirements and preferences.

To address these challenges, my research focuses on developing frameworks and solutions that improve the performance, scalability, and efficiency of deep learning systems. Specifically, I propose SwitchFlow, a scheduling framework for deep learning multitasking that addresses the limitations of current deep learning frameworks in supporting GPU sharing. I also propose Atom, a distributed training framework for large language models that utilizes decentralized training to reduce communication costs and increase scalability. In addition, I present Perfect, a method to improve the data processing efficiency by first pre-training repetitive data and then fine-tuning the model to the target accuracy.

By proposing these frameworks and solutions, I aim to contribute to the advancement of deep learning systems, which have the potential to significantly impact a wide range of applications. My research seeks to improve the efficiency, scalability, and performance of deep learning systems, ultimately enabling the development and deployment of more complex and accurate deep learning models.

1.2 Challenges in Deep Learning Systems

Deep learning systems are critical for addressing the unique challenges posed by deep learning applications.

Dealing with massive amounts of data can pose a significant challenge in the context of data preprocessing. This process is crucial for preparing the data to be suitable for training deep learning models, which typically require large quantities of high-quality data. Without efficient data preprocessing, the time and resources required to clean, transform, and augment the data can become a bottleneck in the model training process. Therefore, optimizing the data preprocessing pipeline is essential to maximize the performance of deep learning models on large datasets.

Model training, which involves optimizing the parameters of the deep learning model using large datasets, requires powerful computing resources and distributed computing techniques to achieve acceptable training times. Especially, for large language models (LLMs), the resource requirements for model training can be even more demanding. One promising solution to address these challenges is decentralized training. However, this approach also brings its own set of challenges that need to be overcome. In the thesis, I explore these challenges and propose potential solutions to address them.

Resource management is crucial for allocating and managing computational resources efficiently, especially in multi-tenancy settings, to avoid bottlenecks and ensure smooth execution of training and serving tasks. Multi-tenancy is crucial for supporting concurrent usage of deep learning systems by multiple users or applications, enabling efficient sharing of resources and ensuring fair allocation. Monitoring and runtime system are essential for tracking the progress of model training, identifying and resolving issues, and ensuring smooth deployment and serving of deep learning models in production.

1.3 Outlook: Towards Optimizing Resource Utilization, Efficiency and Scalability in Deep Learning Systems

Deep learning systems are essential for achieving high-performance deep learning training and serving, and they play a critical role in enabling state-of-the-art results across various domains. My research focuses on understanding and developing deep learning systems that encompass infrastructure, data preprocessing, ML code, model training, distributed computing, resource management, monitoring, runtime systems, and multi-tenancy. Through my work, I aim to advance deep learning systems and their impact on deep learning applications.

My research aims to provide valuable insights into state-of-the-art deep learning systems and their relationship with deep learning applications, paving the way for more efficient and scalable solutions in the field of artificial intelligence. I showcase how advancements in deep learning systems directly impact the performance, scalability, and efficiency of deep learning applications. By addressing the challenges and advancements in deep learning systems, my research contributes to the field of deep learning and accelerates the development of innovative and high-performance deep learning applications.

The significance of deep learning systems in enabling the success of deep learning applications cannot be overstated, given the increasing demand for these applications across various domains. In the following sections, I delve into the details of deep learning systems, including their components, challenges, and state-of-the-art solutions.

1.4 Thesis organization

The remainder of this thesis will be structured as follows:

Chapter 2 delves into the design and implementation of SwitchFlow, a scheduling framework for deep learning multitasking. The chapter first highlights the demand and challenges of effectively sharing GPU resources among multiple DL models. It then presents the current design limitations of popular DL frameworks, such as TensorFlow and PyTorch, in supporting GPU sharing. The proposed SwitchFlow framework centers on two designs: subgraph scheduling and subgraph versioning. The former prevents interference among subgraphs from different models running on a single GPU, while the latter enables low-latency preemption by allowing subgraphs to migrate across devices at a low cost. The chapter concludes by presenting the results of representative DL models that demonstrate the effectiveness of SwitchFlow in achieving up to an order of magnitude lower tail latency for inference requests collocated with a training job. The material presented in this chapter is based on joint work with Jia Rao, Wei Chen, Hang Huang, Chris Ding, and Heng Huang [129].

Chapter 3 introduces Atom, an elastic, fault-tolerant distributed training framework designed to support the training of massive-scale language models on more affordable hardware resources. The chapter first discusses the challenges that arise when training large-scale models due to the lack of specialized hardware resources. It then presents the proposed Atom framework, which utilizes model swapping and parallel training of multiple copies to achieve high training throughput. Unlike existing model partitioning approaches, Atom aims to accommodate an entire LLM on a single host and uses static analysis to determine an optimal model partitioning scheme and schedule that seamlessly overlaps model execution with swapping. The chapter also highlights the advantages of Atom, including its elimination of the single point of failure in pipeline parallelism approaches and its superior performance and scalability compared to tightly-coupled pipeline parallelism with low-speed networks. The chapter concludes by presenting the experimental results with various configurations of the GPT-3 model that demonstrate the effectiveness of Atom in improving training performance by up to 20× for poor network connections compared to state-of-the-art decentralized pipeline parallelism approaches. The material presented in this chapter is based on joint work with Jia Rao, and Wei Chen.

Chapter 4 presents *PerFect*, a novel method for improving the throughput performance of the data preprocessing stage in deep learning pipelines. This chapter discusses the limitations of existing solutions, including data echoing and specialized data loading libraries, and proposes a new approach that avoids these limitations without the need for additional hardware or third-party libraries. The chapter provides theoretical insights and evaluates the proposed method on CIFAR10 and ImageNet datasets using various models. The results of comprehensive experiments demonstrate the efficacy of *PerFect* in achieving the target accuracy while reducing training time. The material presented in this chapter is based on joint work with Jia Rao.

Overall, this thesis investigates challenges in deep learning pipelines and presents innovative solutions to improve their performance. The following chapters cover topics such as preemptive multitasking, distributed training, and data preprocessing, each providing new insights and techniques to improve the efficiency and scalability of deep learning models.

Chapter 2

SwitchFlow: Preemptive Multitasking for Deep Learning

Accelerators, such as GPU, are a scarce resource in deep learning (DL). Effectively and efficiently sharing GPU leads to improved hardware utilization as well as user experiences, who may need to wait for hours to access GPU before a long training job is done. Spatial and temporal multitasking on GPU have been studied in the literature, but popular deep learning frameworks, such as TensorFlow and PyTorch, lack the support of GPU sharing among multiple DL models, which are typically represented as computation graphs, heavily optimized by underlying DL libraries, and run on a complex pipeline spanning CPU and GPU. Our study shows that GPU kernels, spawned from computation graphs, can barely execute simultaneously on a single GPU and time slicing may lead to low GPU utilization.

This section presents SwitchFlow, a scheduling framework for DL multitasking. It centers on two designs. First, instead of scheduling a computation graph as a whole, SwitchFlow schedules its subgraphs and prevents subgraphs from different models to run simultaneously on a GPU. This results in less interference and the elimination of out-of-memory errors. Moreover, subgraphs running on different devices can overlap with each other, leading to a more efficient execution pipeline. Second, SwitchFlow maintains multiple versions of each subgraph. This allows subgraphs to be migrated across devices at a low cost, thereby enabling low-latency preemption. Results on representative DL models show that SwitchFlow achieves up to an order of magnitude lower tail latency for inference requests collocated with a training job.

2.1 Introduction

Recent advances in deep learning (DL) [65] have led to the wide adoption of machine learning techniques in image classification [50, 65], speech recognition [49], and natural language processing [34]. The success of deep learning can be partially attributed to the enormous amount of data available for model training and the advent of fast graphics processing units (GPUs) that allows more sophisticated models (e.g., deep neural networks (DNNs)) to be trained at a speed an order of magnitude faster than that on CPUs. The surge of deep learning has also given rise to deep learning frameworks, such as TensorFlow [6] and PyTorch [93], which make programming complex models not only easier but also more efficient on various accelerators (e.g., GPUs, TPUs [61], and FPGAs).

As deep learning continues to gain popularity and is increasingly deployed in cloud services [88, 67], there is a growing need for sharing accelerators [12, 128, 90, 14, 125, 81, 84] (e.g., GPUs) among multiple deep learning workloads. Multitasking has been a key feature in modern computing systems to share a single device. *Spatial multitasking* partitions resources among multiple tasks and executes them simultaneously if their combined size can fit in the device. *Temporal multitasking* assigns each task a time quantum during which the device is dedicated to one task at a time. Both mechanisms are proven effective for improving GPU utilization [7, 90, 57, 131, 14]. However, multitasking deep learning workloads, which are inherently more complex than simple GPU kernels, presents unique challenges.

First, DNNs written with DL frameworks contain complex execution flows, typically in the form of a computation graph with thousands of nodes. Each node in the graph is a mathematical operation to be executed on either CPU or GPU. Multitasking deep learning workloads requires that the scheduling of the CPU and GPU nodes (kernels) in the same graph be coordinated. The existing architectural support for

GPU multitasking is limited to concurrent execution of independent kernels on GPU hardware, thereby unable to handle the multitasking of complex deep learning models due to the lack of knowledge of computation graphs from multiple users. There are existing works that control the launching of GPU kernels in the runtime to enable GPU sharing [128, 24]. However, switching computation graphs (i.e., DL models) is non-trivial. DL frameworks support two graph execution modes. 1) *Dynamic graph* mode, the default execution mode in PyTorch and TensorFlow (also known as eager execution), generates graph nodes on-the-fly as model execution proceeds. It allows for evaluating the output of graph nodes immediately after its execution, thereby offering an intuitive programming interface and facilitating debugging. 2) In contrast, *static graph* mode builds the entire graph before model execution and performs offline graph optimizations. The resulted graph allows for node merging, reordering, and concurrent node execution and is significantly faster than dynamic graphs, especially for large models. Multitasking models with static graphs is much more challenging because node execution does not follow user's code and is asynchronous and interleaved.

Pipeswitch [14] leverages **dynamic graphs** in PyTorch to enable fast model switching via pipelined model transmission. It relies on layer-by-layer model execution in dynamic graphs to overlap model transmission and execution, which is critical to efficient DL multitasking. In this chapter, we take the challenge to support DL multitasking on **static graphs**, which are widely adopted in production systems due to high performance, efficient computation graph optimization, ease of co-design of hardware acceleration and compiler optimization.

Second, DL frameworks, such as TensorFlow (TF), rely heavily on machine learning libraries, e.g., NVIDIA cuBLAS and cuDNN [83, 27] to accelerate frequently used routines in DNNs, such as convolution and matrix multiplication. The DL libraries carefully tune GPU kernels based on GPU resource availability, such as the number of streaming multiprocessors (SMs), cores per SM, and the size of device memory. Since there lack mechanisms for dynamically reconfiguring GPU resources, the tuning must be performed before model execution. Therefore, users need to explicitly set resource limits, e.g., memory size, for each DL model. This requires that either DL models be allocated with statically partitioned resources to allow concurrent model execution or the entire GPU should be allocated to one model and models have to be executed one after another. While TF allows for dynamic memory growth, which allocates GPU memory only when models actually use it, TF does not support reclaiming GPU memory until model execution is completed. Thus, it is not suitable for DL multitasking. Recent work AntMan [131] realizes elastic memory management for DL models based on GPU unified memory [80] and allows model data to be freely allocated on both GPU and host memory. However, AntMan does not address job preemption and can only switch DL jobs at the completion of mini-batches. In practice, DNN training jobs are usually allocated dedicated GPUs [58, 44, 2] while multiple inference jobs may be packed on a single GPU [88]. As a result, training jobs cannot share a GPU for lack of memory and inference jobs may experience high latency waiting for training jobs to complete due to the lack of an effective preemption mechanism.

Third, new challenges and opportunities arise surrounding multitasking DL workloads: 1) Like in conventional workload collocation, DL multitasking should meet different service-level objectives (SLOs) for heterogeneous workloads. While model training is throughput oriented and requires high resource utilization, model serving (i.e., inference) has stringent latency requirements. However, model training is significantly more resource-intensive than inference, not only requiring an order of magnitude more GPU memory but also computing power for model parameter updates (i.e., updated weights) through iterations. In addition, inference could experience high latency due to the long preemption latency of training. 2) New multitasking scenarios emerge as DL continues to evolve [75]. Multi-task learning [104] trains multiple models from the same training data set. For example, separate models should be trained to detect pedestrians and vehicles, respectively, from the same set of sensing data in autonomous driving. Since these DL models share the input and possibly some layers of a DNN, running them on the same GPU opens up new opportunities for exploiting data locality. During model training, a large set of training samples is usually divided into mini-batches, each of which can fit in GPU memory. The existing DL frameworks repeatedly load mini-batches into GPU for training separate models even though each mini-batch can be shared among models. This motivates the development of a new GPU multitasking scheme that allows for fine-grained data reuse on GPU across different models.

In this chapter, we present SwitchFlow, a scheduling framework for multitasking DL jobs. We identify **several issues** in static computation graph execution in TensorFlow, the arguably best-performing DL framework for production systems. **First**, computation graph execution typically employs multiple worker threads to exploit concurrency in executing computation graph nodes. There is a lack of an effective and efficient preemption mechanism to enforce priority between different graphs from multiple jobs in static execution mode. **Second**, our empirical study revealed that DL operations, which are optimized and automatically configured by DL libraries, barely can simultaneously execute on a single GPU, though there is ample concurrency in the computation graph. **Third**, graph execution is a complex pipeline spanning CPU and GPU. DL multitasking should efficiently utilize the heterogeneous devices. Existing work such as session-based time slicing [130, 52], which allows models to exclusively access both CPU and GPU and runs them iteration by iteration, leads to low GPU utilization because GPU waits for CPU to feed input data [76] during each iteration.

SwitchFlow addresses these issues through two designs. **First**, SwitchFlow maintains multiple versions of a computation graph, which includes subgraphs that run on different devices. Replicated subgraphs, each individually optimized for different devices (CPU or GPU) for the same computation, enable SwitchFlow to freely migrate the execution of subgraphs between CPU and GPU and vice versa. **Second**, unlike in TensorFlow, wherein nodes in a computation graph are indistinguishably scheduled by workers, SwitchFlow treats nodes to be executed on different devices, e.g., CPU or GPU, differently in scheduling, following two principles: 1) GPU nodes from different models are not scheduled simultaneously on a single GPU, allowing exclusive access to GPU; 2) all other nodes, including CPU nodes and GPU nodes on a different GPU, are allowed to run without restrictions to improve pipeline efficiency.

The result is a design that allows users to provision GPU resources for their models without concerns about interference from other models or memory over-commitment. It also enables a low-latency, low-cost preemption mechanism to deschedule an entire computation graph without throughput loss. Experimental results on representative DL models and three different GPUs show that 1) SwitchFlow achieves a 19.05x tail latency improvement for inference requests when collocated with a heavy-weight training job compared to an variant of TF. 2) SwitchFlow is more efficient than the existing time slicing-based approaches in utilizing heterogeneous devices. 3) With user-provided hints, SwitchFlow is able to merge multiple computation graphs of similar models to share the data preprocessing stage and achieves up to 65% performance improvement compared to time slicing in multi-task learning.

2.2 Background and Motivation

In machine learning frameworks, such as TensorFlow, learning algorithms are represented as *computation graphs* wherein nodes describe operations while edges specify dataflows between those operations. Expressing machine learning models as computation graphs offers several benefits. First, the execution of a learning algorithm can be accelerated by optimizing the directed graph, e.g., pruning, merging, and partitioning. Second, the abstract representation of computation allows operations to be individually implemented using different machine learning libraries, making them portable across heterogeneous devices. For example, a graph can be executed entirely on CPU or on a hybrid CPU/GPU system. Third and most importantly, computation graphs specify the order of execution and allow concurrent operations to be scheduled in parallel.

In what follows, we discuss the challenges of sharing a single GPU among multiple DL models, each with its own computation graph. Without loss of generality, we focus our discussion on the graph execution mode in TensorFlow (TF), which is based on static graphs. Our discussions are based on TensorFlow 1.13.1. TF 2.0 has undergone drastic changes to its runtime to improve programmability. The default eager

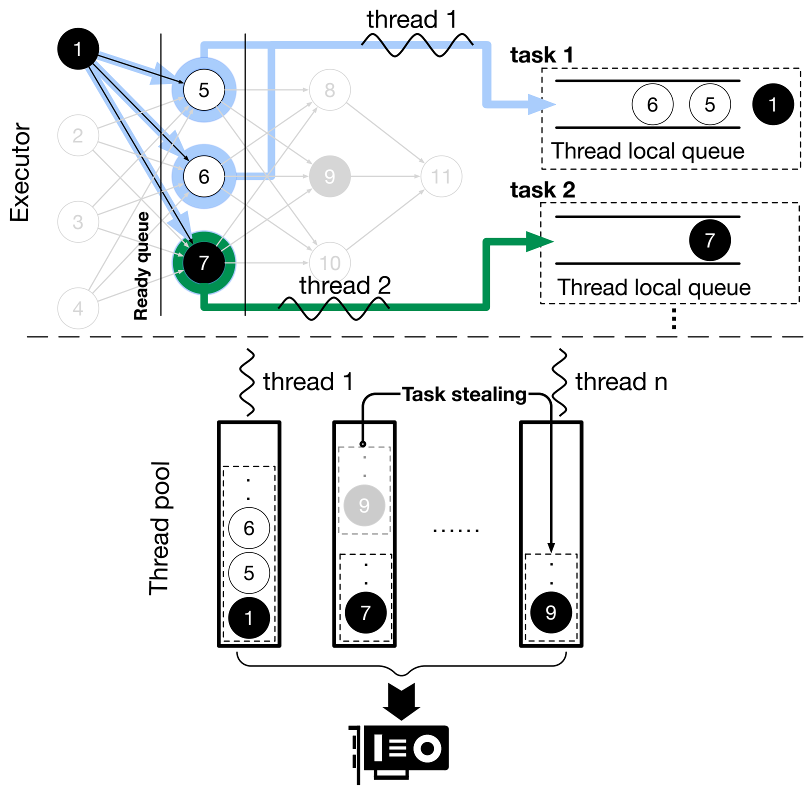


Figure 2-1: computation graph scheduling in TensorFlow.

execution mode in TF 2.0 does not use computation graphs and thus is considerably slower due to the lack of parallelism in model execution.

2.2.1 Executing Computation Graphs in TF

To execute a computation graph, resources need to be provisioned for graph nodes, i.e., operators (*ops*), and a schedule plan needs to be determined to run them on different devices, e.g., CPUs and GPUs, while enforcing node dependency. TF's graph execution centers on three techniques: *session*, *executor*, and *thread pool.*

Session is a runtime instance created by users to execute graph nodes associated with an output node. The target output can be an intermediate node or the final node of a graph. In the former case a subgraph is executed while in the latter the full graph is executed. In deep learning, a `session.run` performs one iteration of training or inference. For training, the parameters (weights) of the model are updated after each session run. During session construction, different devices (e.g., CPU and GPU) are added to a session and a cost model is used to determine the backend device to execute each node. TF makes use of external, highly-optimized numerical libraries, such as MKL [4], cuBLAS [83], and cuDNN [27], to implement operations (kernels) on CPU or GPU. Session also optimizes graph execution by partitioning the full graph into subgraphs, which can be independently executed by *executors*.

Executor dispatches operations from a subgraph to several task queues from where they are executed by worker threads in a **thread pool**. Figure 2-1 shows how operations are scheduled to run in an executor. Note that there could be multiple executors in a session, each including nodes to be executed on a single device. For example, in a 2-GPU system, there are typically three executors, one for operations running on CPU and one for each of the two GPUs. Executor uses the input size and the type of operation to determine the cost of each node and classifies them into expensive and inexpensive ops. There is a single *ready queue* for each executor wherein nodes in the subgraph are inserted in a breadth-first manner. Initially, all nodes (expensive or inexpensive) in the ready queue are concurrent and dispatched to separate *local queues*, each of which will be processed by a worker from the thread pool. Workers launch ops (kernels) from their local queues in FIFO order. After a node is done, its subsequent nodes which have a direct edge from the current node are inserted in the ready queue. Unlike in the initial dispatch, only expensive nodes require to be placed on a new local queue and inexpensive nodes are sent to the local queues of their parent node. Each local queue is assigned to a worker for node processing and a thread is put to sleep if its queue is empty. Before sleep, a thread performs random work stealing from other threads to balance the load.

2.2.2 Challenges in Multitasking DL Workloads

Executor-based computation graph scheduling exposes ample concurrency to build an efficient execution pipeline for DL models: 1) Since executors are associated with different devices, their computation is independent from each other and can be done in parallel except for cross-device data transfer. Therefore, stages (subgraphs) for reading input data and preprocessing, which are typically done on CPU, can overlap with training or inference on GPU. 2) Concurrent nodes in a subgraph are processed by multiple workers in parallel. However, expressing and executing DL algorithms using computation graphs present great challenges in sharing computing systems among multiple DL workloads.

Task preemption is the mechanism to suspend the currently executing task, save its states, and switch to another task. It is crucial to enable differentiation and time sharing among tasks. However, there is no effective and efficient preemption mechanism for DL workloads. Due to the parallelism in computation graphs, there could be multiple GPU kernels of the same DL model simultaneously running on GPU or waiting at the launch queue. Additionally, multiple CPU threads processing the CPU executor can run asynchronously with their GPU counterparts on multiple CPUs. To preempt a DL workload, all the three types of tasks associated with the DL model should be suspended and their contexts be saved. The existing hardware mechanisms for GPU context switching [91] lack the knowledge of computation graphs and are limited to preempting a single GPU kernel (node).

Furthermore, DL operations, such as those in DNN training, are memory intensive. The intermediate data generated during model training, e.g., gradients, could be an order of magnitude larger than the input [102, 72]. For a mini-batch of 64-128 images, its input size ranges from tens to hundreds of megabytes (MBs). Suspending tasks of a DL model during training requires to save a context of a few to tens of gigabytes of data, leading to not only high storage cost but also long context saving time. Alternatively, preemption can happen until an iteration (a session in TF) is finished so that only the data that should persist across iterations, such as model weights, is saved. However, as reported in [130] and verified by our experiments, an iteration of training can take up to 1s and add a sizable delay to preemption. Long preemption latency is not acceptable for latency-sensitive DL inference workloads [32, 88, 52, 67].

As DL workloads are executed on complex pipelines across multiple devices, e.g., CPU and GPU, it is non-trivial to efficiently share heterogeneous devices among DL workloads. While there have been extensive studies on spatial [7] and temporal multitasking [90] on GPUs, we demonstrate that DL operations can hardly execute

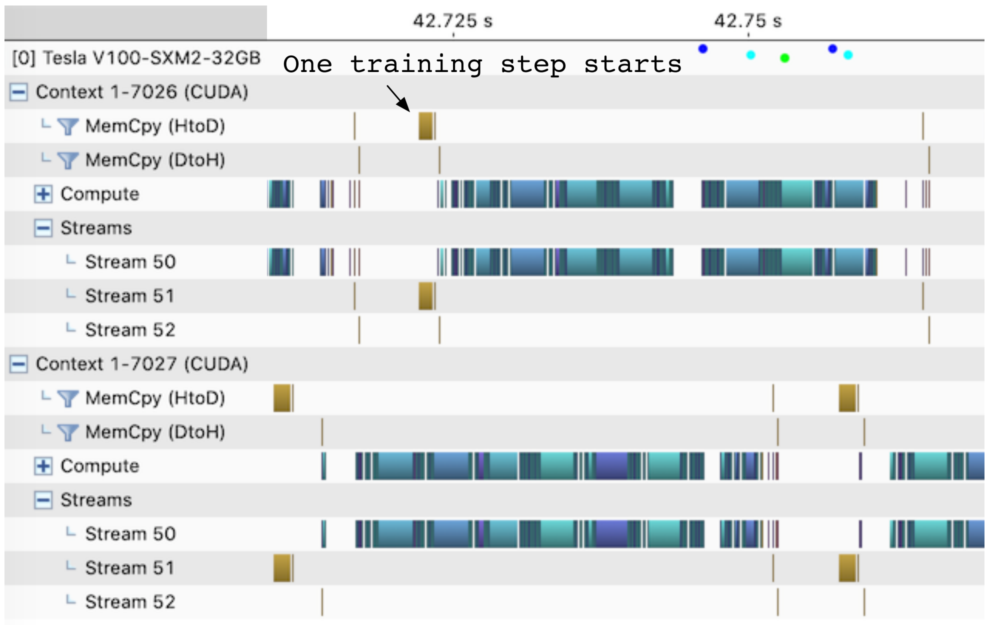


Figure 2-2: The timeline of training two ResNet50 sharing a single NVIDIA V100 GPU.

simultaneously on GPU and time slicing GPU can lead to low efficiency.

Ineffective spatial multitasking. Modern GPUs support concurrent execution of several small kernels to improve device utilization. For example, with the help of the Hyper-Q technology [20], NVIDIA multi-process service (MPS) [84] and CUDA streams allow multiple kernels to be launched to multiple hardware work queues. If the kernels are truly independent and their aggregate resource demand fits in the GPU, they can be simultaneously executed. However, spatial multitasking is not effective for DL workloads.

We executed two 2D convolution (`tf.nn.conv2d`) operations, a commonly used routine in DNNs, from two CUDA streams and compared their execution with one stream on a single NVIDIA GPU. We used `nvprof` [85] to collect statistics of primitive routines, such as the block size, number of registers, and used shared memory. Concurrent kernel launch from two streams does not offer much performance benefit. The completion time of two streams is close to executing the two operations sequentially. An analysis of the limiting factors in the kernels using NVIDIA's GPU occupancy calculator [87] revealed that 10 of the 13 kernels were bottlenecked by GPU register files and cannot run concurrently.

We further train two ResNet50 models concurrently on a single NVIDIA V100 GPU with a batch size of 16 on the ImageNet dataset. Figure 2-2 shows the execution timeline of the two models. The color areas are kernel execution on GPU and white areas are the time spent on CPU. We made two observations: 1) while some kernels of the two model can be simultaneously executed on GPU, their execution times were significantly prolonged due to contentions on shared GPU resources. 2) There still exists significant serialization on GPU between the two models in which kernels from one model exclusively occupied GPU while those from the other model were waiting to be issued by CPU. As a result, the training throughput of individual models dropped from 226 to 116 images per second due to GPU sharing. It suggests that spatial multiplexing is barely beneficial.

The reason no two heavy kernels can execute simultaneously without performance loss even with multi streams in the GPU is that primitive routines in cuDNN or cuBLAS are optimized to fully utilize resources on GPU. Although modern GPUs support resource partitioning and library can adapt kernels to meet the constrains, allocation needs to be done when the computation graph is constructed. As workloads in shared systems are dynamic, static resource partitioning likely leads to underutilization. Alternatively, it is possible to control the resource demand of DNNs during runtime without static resource partitioning. By changing the input size, e.g., batch size, the size of kernels can be dynamically adjusted to fit in GPU. However, changing the batch size may lead to longer training time, negating the benefit of resource sharing. Furthermore, the memory demand of individual kernels need to be carefully controlled not to exceed the capacity of GPU memory. Otherwise, DL jobs may crash due to out-of-memory (OOM) errors.

Inefficient temporal multitasking. Time-slicing GPUs among multiple DL workloads has been explored to provide early feedback in training [130] and better quality-of-service (QoS) for inference [52]. For time sharing, computation graphs are switched at the end of a session. Therefore, during a time slice (consisting of one or more sessions), only the nodes from one graph are executed and GPU is dedicated to one

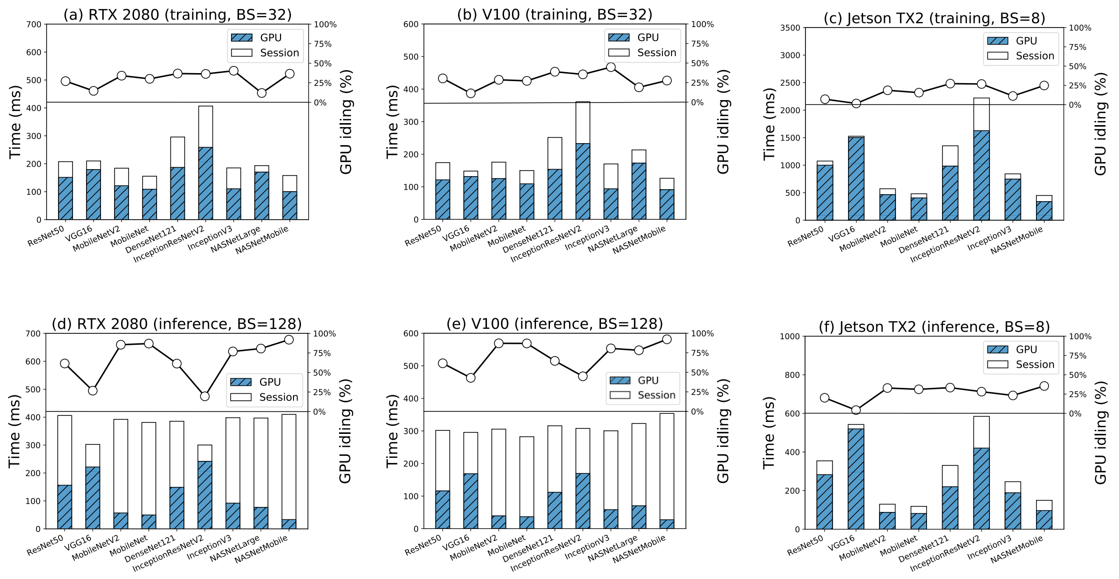


Figure 2-3: The percentage of GPU idle periods due to inefficient DL execution pipeline on three different types of GPU.

DL workload. However, the DL execution pipeline comprises stages on CPU and GPU. As GPUs continue to improve, the early stages on CPU for data loading and preprocessing will increasingly become the bottleneck [28]. Session-based time slicing dedicates the entire pipeline (both GPU and CPU) to one DL job. If the job cannot efficiently utilize the heterogeneous resources, devices may be left idle.

To demonstrate the severity of pipeline inefficiency, we measured GPU idling periods during training and inference on three NVIDIA GPUs: a cost-effective GPU (GeForce RTX 2080 Ti), a high-end GPU (V100), and a power-efficient embedded GPU (Jetson TX2). We used the TF *timeline* profiler [46] to measure the GPU busy time in a session and the length of the session. The difference between the two is the GPU idling period. Figure 2-3 shows the execution time breakdown of 9 CNN models. The measurements were the average of 200 sessions in each model. The white area above GPU time refers to GPU idling. The input was randomly selected images in JPEG format from the ImageNet dataset. Images were grouped into batches to improve GPU utilization. We chose the commonly used batch sizes for training (32) and inference (128). `tf.data.Dataset` was used for data preprocessing. Input prefetching and parallel data workers (32 preprocessing threads) were enabled.

As shown in Figure 2-3 (d) and (e), most models caused long GPU idle periods when performing inference. For instance, in model *NASNetMobile*, for more than 90% of the time, the V100 GPU was idle waiting for CPU to feed data. In contrast, the computation on CPU and GPU can be better overlapped in training, as shown in Figure 2-3 (a) and (b), because training includes one forward and one backward pass in each session and requires more GPU computation. For the embedded GPU (TX2), GPU was the bottleneck in most models for both training and inference. We also observed two trends. First, faster GPU results in more GPU idling. Second, increasing the batch size leads to more GPU compute time but will further exacerbate GPU underutilization in a session as data preprocessing becomes even longer.

Summary. We have shown the difficulties in running multiple DL jobs on a GPU simultaneously and the low efficiency of GPU time slicing. This motivated us to develop a more flexible and efficient approach for DL multitasking.

2.3 SwitchFlow: System Design

2.3.1 Overview

Deep learning workloads can be broadly categorized into training and serving. Training workloads are throughput-oriented, computationally expensive, and long-term. By contrast, serving workloads have a tight latency requirement but execute for the short term, leading to low utilization in a production environment since online inference queries often arrive unpredictably and stochastically.

The major problem of session-based computation graph execution is the coupling of stages (executors) running on heterogeneous devices. Simultaneously running multiple sessions causes contention on bottleneck devices, making it hard to guarantee QoS and even resulting in OOM crashes due to memory overcommitment. On the other hand, restricting only one session to access computational resources (time slicing) leads to low utilization. In contrary, SwitchFlow views computation graphs as a

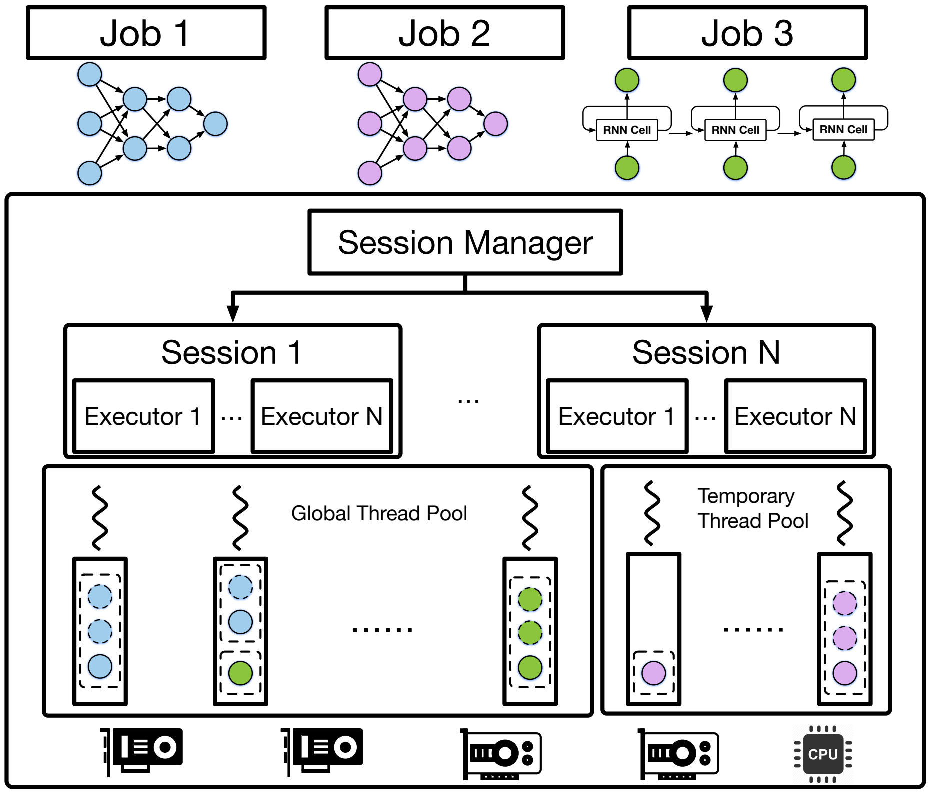


Figure 2-4: Design of SwitchFlow.

set of executors that can be flexibly managed and scheduled across sessions.

This design offers several benefits: 1) By replicating executors for each available device during graph construction, the execution of graphs can be timely suspended and migrated, enabling low-latency, low-cost DL job preemption. 2) Executors from similar jobs can be assembled to exploit data reuse in multi-task learning. 3) Executors of different types and from different jobs can be interleaved to efficiently utilize heterogeneous resources.

2.3.2 Session Management

The central idea in SwitchFlow session management is allowing sessions from any DL jobs to access all available devices on a machine. Unlike TF, in which sessions are statically configured with a fixed number of devices and each session has its own thread pool for graph execution, SwitchFlow shares all devices and a single global thread pool among sessions, as shown in Figure 2-4. The temporary thread pool is used for fast preemption and will be discussed in Section 2.3.3. Furthermore, SwitchFlow creates multiple executors, each corresponding to an available device on the machine, for each subgraph during graph construction. Initially, for a subgraph, the executor and its associated backend device are determined by the ML framework based on a cost model. The additional copies of executors are used for migrating a subgraph from one device to another.

The session manager determines when to schedule executors from sessions while preserving the dependency within a session. It allows executors from different sessions to interleave but ensures that executors from consecutive runs of the same session follow sequential order. SwitchFlow supports independent DL jobs as well as multi-job scenarios, where a user runs a set of jobs on the same training set to tune hyper-parameters, such as the number of layers/weights, mini-batch size, and learning rate, of a model [130] or to train multiple models. In multi-job scenarios, SwitchFlow merges subgraphs from different but correlated sessions based on user-provided configuration.

2.3.3 Preemption

SwitchFlow addresses several challenges in task preemption. First, to preempt a DL job, all its tasks queued in the ready queue, local thread queues, dispatched onto GPUs, and currently running on CPUs must be stopped in a coordinated and timely manner. Second, the context of the suspended job must be saved and the storage cost should be contained. Third, task resumption should also be fast to avoid throughput loss.

Recall from previous discussions that worker threads independently dispatch tasks and can steal from each other. It is necessary to isolate high priority jobs (pre-empters) from those to be preempted (preemptees). As shown in Figure 2-4, SwitchFlow maintains a temporary thread pool to handle the preemptees until preemption is completed. With the help of the executor scheduler (discussed in Section 2.3.4), SwitchFlow guarantees low preemption latency.

Task suspension. Upon the arrival of a high priority DL job, the session manager first aborts the nodes that are currently queued in the ready queue and thread local queues from the preempted job. The kernels that have been dispatched onto GPU are allowed to finish as they may be interleaved with other jobs' launched kernels and there is a lack of mechanisms to selectively stop kernels of a particular job. Second, the session manager reconstructs the computation graph of the preempted job to replace the executor (subgraph) on the bottleneck device with an alternative executor on a different device. For example, if *job1* is to preempt *job2* on GPU1, *job2*'s executor on GPU1 will be replaced with an executor on GPU2 or CPU. As such, subsequent sessions of *job2* will be run on a different device, isolated from the high priority job. Furthermore, subsequent sessions of the preempted job will be handled by the temporary thread pool until preemption is completed. This ensures that the launching of the new job is not interfered.

After preemption is done, the preempted job can be moved back to the global thread pool. In the case that there is no GPU available and the preempted job has to run on CPU, e.g., using an executor implemented with the Intel MKL library, we keep it in the temporary pool to prevent a large number of MKL operations from exhausting the global thread pool. At initialization, SwtichFlow spawns as many threads as the number of cores in each thread pool and uses wakeup signals to control the number of active threads. Thread count in the temporary pool can be configured by configuration and is a tradeoff between isolation and the performance of preempted jobs. SwitchFlow ensures that the total number of workers in the two thread pools matches the number of cores.

Context saving and task resumption. For training jobs, model weights that persist across iterations (i.e., session runs) need to be saved to preserve training progress. For inference, no cross-session state needs to be saved since prediction requests do not update model weights and are independent. ML frameworks keep model weights in GPU memory across iterations and copy updated weights back to host memory after training is completed. At the end of each iteration, intermediate data, such as calculated gradients, is discarded but weights remain in GPU memory. To save job context, SwitchFlow tracks persistent variables in a session through TF's resource manager on each device. To reduce the delay caused by state transfer, SwitchFlow does not initiate the transfer when preemption is in progress, i.e., the session of a preempted job is being aborted. Instead, SwitchFlow waits until a new session run of the preempted job is started.

A preempted job is migrated to a different device and allowed to resume immediately. The session manager uses the newly constructed graph to start a new session run of the preempted job. The new session is populated with the tasks of the aborted session run so that no work is lost. Most importantly, before the job is resumed, SwitchFlow copies the model weights from the GPU where the job is preempted to the new device using asynchronous memory copy. Note that the state transfer is off the critical path of preemption and can be performed concurrently with the high priority job. However, this requires model weights to be preserved on the source GPU until state transfer finishes, occupying GPU memory that can be used by the new job. We think this is a necessary tradeoff for minimizing preemption latency. As will be shown in Section 2.5.2, intermediate data dominates model memory usage [102, 72] and weights only account for less than 10% of the total memory usage.

2.3.4 Scheduling

Recall the two issues of computation graph execution: 1) primitive routines (kernels) are highly optimized by DL libraries to improve hardware efficiency, thereby unable to co-run on a single GPU without performance loss; 2) the execution pipeline of a single model cannot efficiently utilize both CPU and GPU. To this end, SwitchFlow maintains two **scheduling invariants**:

First, no two GPU executors are scheduled on a single GPU simultaneously. It is worth noting that this constraint not only helps more efficiently utilize GPU through time slicing but also effectively avoids OOM errors as well as offering flexibility for resource provisioning. Users can assume they have exclusive access to GPUs they request and configure their models accordingly, e.g., selecting an appropriate batch size. Since model weights need to be preserved on GPU, the aggregate size of persistent variables of all models sharing the same GPU should not exceed GPU memory size. SwitchFlow allows one GPU executor to finish before switching to another. As

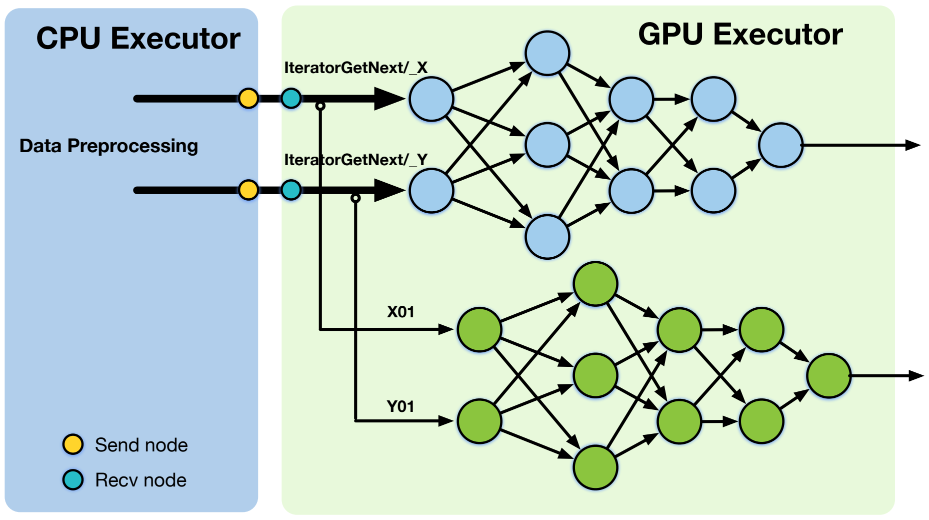


Figure 2-5: Input data reuse in multi-task learning.

such, intermediate data is discarded and a majority of GPU memory is freed.

We use preemption to demonstrate how this scheduling constraint achieves low preemption latency without OOM errors. A high priority job is allowed to start immediately after submission. The new job goes through computation graph construction before its executors are ready to run. If the preempted job is still being aborted, its GPU executor is running and will prevent the new job's GPU executor from starting. Therefore, the abort time and preparation time can be overlapped.

Second, executors on different devices can be scheduled freely. SwitchFlow does not impose restrictions on the scheduling of CPU executors or executors on different GPUs. This is in stark contrast to session-based time slicing wherein no executors from other sessions can be scheduled. SwitchFlow allows any CPU executor to run while a GPU is occupied. It helps to overlap data preprocessing on CPU in one job with GPU processing in another job. Note that we did not observe much contention on CPU because when one job's GPU executor runs, its CPU executor only prefetches input for the next session without processing them. Therefore, CPU executors may not reach their peak demands at the same time.

SwitchFlow supports **customized scheduling** as directed by user configuration. In multi-task learning, users use the same input to train or perform inference on multiple models. During hyperparameter tuning, the same input data is used to navigate the hyperparameter space, e.g., learning rate, momentum, and dropout rate, on the same model. For these use cases, a straightforward approach is to replicate the input and run multiple jobs separately. Not only will it lead to redundant data preprosessing but it also results in low pipeline efficiency. Research in DL showed that multi-task learning can be achieved by sharing the hidden layers of a neural network among multiple models while keeping model-specific output layers [23]. However, the internal structure of these models must be similar and they have to be deployed together.

As shown in Figure 2-5, SwitchFlow offers an alternative way to jobs with same input pipeline. It merges multiple computation graphs to share the data preprocessing stage. Specifically, the `recv` nodes on GPU executors are linked together to share the tensors received from the CPU executor. Note that the input tensor may be modified during GPU processing and is deallocated after the GPU executor finishes, SwitchFlow maintains an immutable copy of the tensor in GPU global memory and makes its address public to all GPU executors sharing the tensor. Models are executed in lockstep. All models should finish processing an input tensor before moving onto the next input batch. To this end, SwitchFlow executes a strict schedule: a shared CPU executor for data loading and preprocessing followed by each model's GPU executor in a round-robin manner.

2.4 Implementation

In this work, we have implemented a prototype of SwitchFlow in TensorFlow. We made changes to TF with 3K+ lines of Python and C++ code. Most changes were made to TF's session and executor management as well as the provisioning of the temporary thread pool and implementing preemption. Executor scheduling was implemented in each session by imposing synchronization among GPU executors that share the same GPU. As the number of models sharing a GPU is typically small (2-3), we used atomic instructions to synchronize on a flag and did not observe noticeable scalability issues. Sessions that do not share GPU schedule independently.

Listing 1: The `launch.py` program for sharing the data preprocessing stage between two models.

```python
# Setup
os.environ['TF_SET_REUSE_INPUTS'] = 'True'
os.environ['TF_REUSE_INPUT_OP_NAME_MASTER_X'] = 'X00'
os.environ['TF_REUSE_INPUT_OP_NAME_MASTER_y'] = 'y00'

# For a master and a secondary model(X01,y01)
os.environ['TF_REUSE_INPUT_OPS_NAME_SUB_X'] = 'X01'
os.environ['TF_REUSE_INPUT_OPS_NAME_SUB_y'] = 'y01'

def launch():
  # master graph
  t0 = threading.Thread(
    name='t0', target=user_00.BuildAndRunGraph,
    args=('graph_00', 'X00', 'y00'))

  # secondary graph
  t1 = threading.Thread(
    name='t1', target=user_01.BuildAndRunGraph,
    args=('graph_01', 'X01', 'y01')
```

Since SwitchFlow runs models using one global thread pool, the models need to run within one TF instance (process) as opposed to one model per instance in the vanilla TF. In the prototype, we employed multiple Python threads to launch models from the same TF instance. As such, users' models written in Python need to be converted into modules and imported to a main `launch.py` program. This implementation can be improved to employ the gRPC interface for model submission, in a way similar to TF serving [88].

It is straightforward to adapt Python TF models to run with SwitchFlow. It takes 1 line of code (LOC) to configure priority for model preemption and 4 LOCs to restrict one GPU executor at a time to run on a shared GPU. Listing 1 shows a more sophisticated case to share the input preprocessing stage between two models. Only 5 LOCs need to be added to a launcher program. The required changes are to add environment variables to configure input sharing. Input reuse can be conveniently enabled/disabled (line 1) and models which share the input with a master model link their `recv` nodes on the GPU executor to the `recv` nodes in the master model (line 2-8). The two models are then launched from two Python threads with the shared stage as an argument (line 10-19).

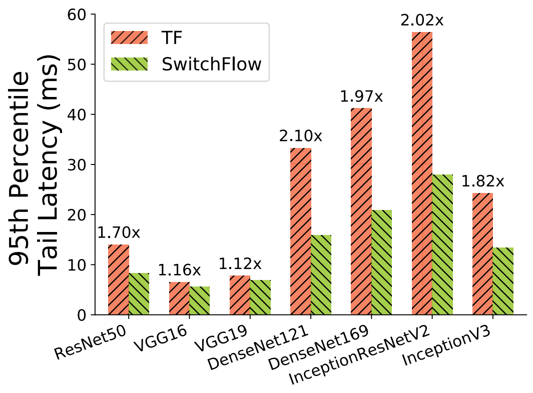
2.5 Evaluation

This section evaluates the effectiveness of SwitchFlow for representative DNNs on four different GPUs. Since TF does not support sharing a GPU, we compare Switch-Flow against two variants of TF: i) multi-threaded TF that uses multiple streams for spatial sharing and ii) TF with session-based time slicing, similar to Gandiva [130]. iii) NVIDIA MPS [84]. Experimental results show that 1) SwitchFlow's preemption mechanism is effective, achieving up to an order of magnitude improvement on prediction tail latency (Section 2.5.2) and maintaining high throughput (Section 2.5.2), 2) Input reuse among correlated models (Section 2.5.3) and interleaved execution of independent models (Section 2.5.4) leads to significant performance improvements in prediction jobs.

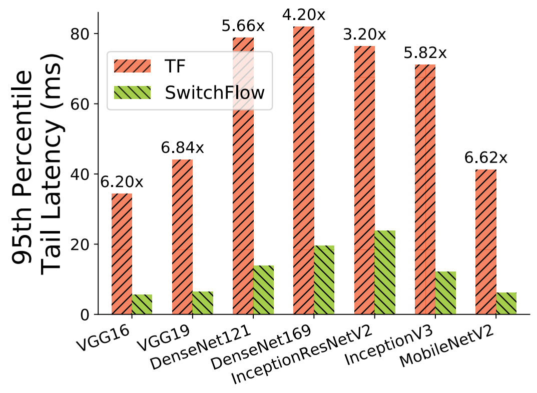
2.5.1 Experimental Setup

Machine configuration. Experiments were conducted on two servers and a Jetson TX2 development kit, all running Ubuntu 16.04. One server was equipped with two different NVIDIA GPUs: GeForce GTX 1080 Ti (11 GB device memory) and RTX 2080 Ti (11 GB) and the other server was with 4 NVIDIA Tesla V100 GPUs (32 GB). Both servers had dual 18-core Intel Xeon processors and over 250GB memory. The CPU and memory performance of the servers is comparable. Jetson TX2 is an embedded computing board with a quad-core ARM Cortex-A57, a 256-core Pascal GPU, and 8GB memory shared between the CPU and GPU.

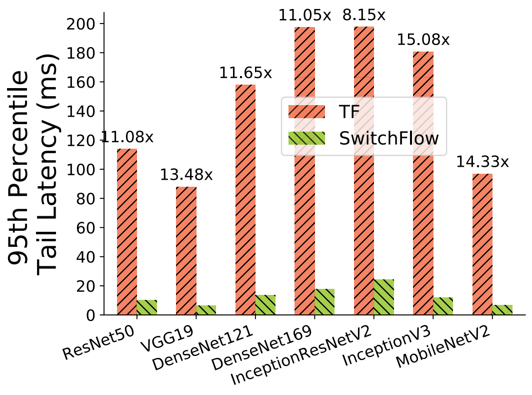
Software. We implemented SwitchFlow on TensorFlow and used variants of TF with the same version for comparison. The CUDA version was v10.0 and the machine learning library used was cuDNN v7.6.4.



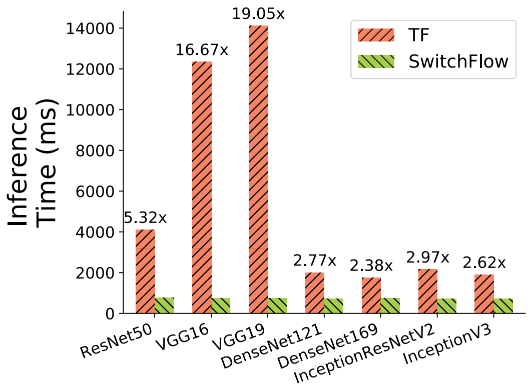
(a) MobileNetV2 (training).



(b) ResNet50 (training).



(c) VGG16 (training).



(d) NMT inference with different CNN training jobs (data series).

Figure 2-6: The 95th percentile tail latency of inference when co-run with background training jobs. The inference request has a higher priority and a batch size of 1. The title of each subfigure shows the background training job.

Benchmarks. Multiple CNN models were selected from Keras applications: ResNet50, VGG16, VGG19, DenseNet121, DenseNet169, InceptionResNetV2, InceptionV3, MobileNet, MobileNetV2, NasNetLarge, NasNetMobile; one recurrent neural network (RNN) model: NMT.

The dataset for CNN models was ImageNet raw JPEG images for evaluation. The dataset for NMT was German-English WMT'16 dataset [3]. For training, the mini-batch size was 32; for inference, if not otherwise stated, batch size (BS) was set to the largest that does not lead to an OOM error.

2.5.2 Effectiveness of Preemption

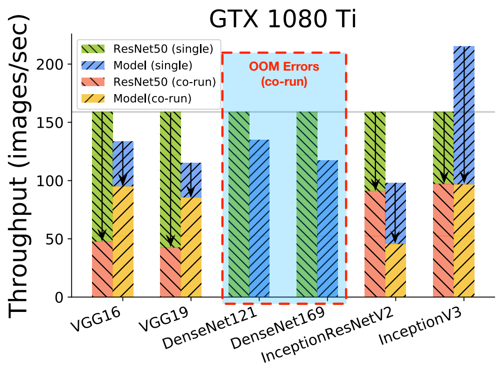
Tail Latency

To evaluate the effectiveness of SwitchFlow's preemption mechanism, we co-ran an inference job with a background compute-intensive training job. Inference requests were configured with higher priority and each contained only one image (BS=1). This ensures that the GPU has sufficient resources to serve the requests and only scheduling affects latency. We first launched the background training job, waited for its warmup, and then submitted inference requests as a continuous stream. The baseline was the multi-threaded TF running training and inference in separate threads, which allowed the two jobs to freely run on GPU.

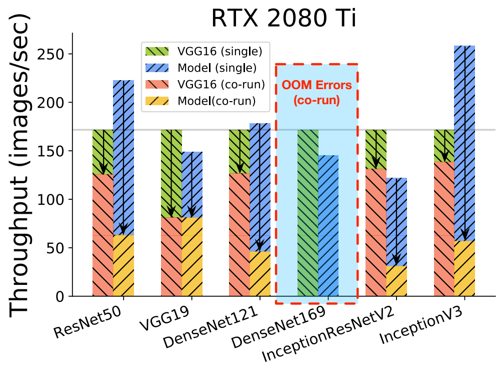
Figure 2-6 shows the 95^{th} tail latency of inference requests due to TF and SwitchFlow. The results show that SwitchFlow achieved significant better tail latency compared to TF. The performance gap varied depending on the resource intensity of the training job. As shown in Figure 2-6 (a)-(c), the performance gap enlarges as models become more computationally expensive. The largest improvement on tail latency (19.05x) was from the test with NMT inference and VGG16 training (Figure 2-6 (d)). RNN inference itself is fairly expensive on GPU and was significantly slowed down when co-running with another expensive model VGG16.

We also evaluated two variants of multi-threaded TF and they incurred even longer delay to inference requests, thereby their results not shown. The first TF variant enforced task priority in the global thread pool. However, since worker threads perform work stealing oblivious of job type, priority inversion occurred and tasks execution of the training and inference jobs were interleaved. The second TF variant employed session-based time slicing and assigned inference a higher priority. Because this approach lacks preemption, in the worst case, inference had to wait for a full session of training to finish.

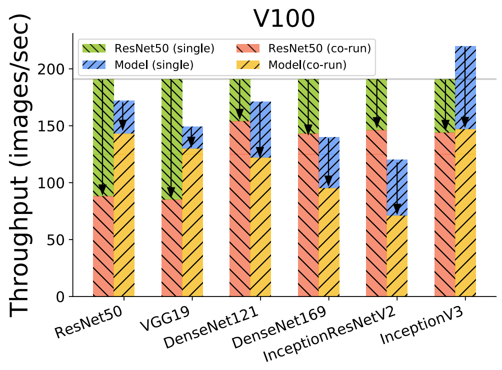
In contrast, SwitchFlow achieved consistently low latency across all workloads. The absolute tail latencies against different models were similar, suggesting that SwitchFlow was able to timely preempt current jobs regardless of their resource inten-



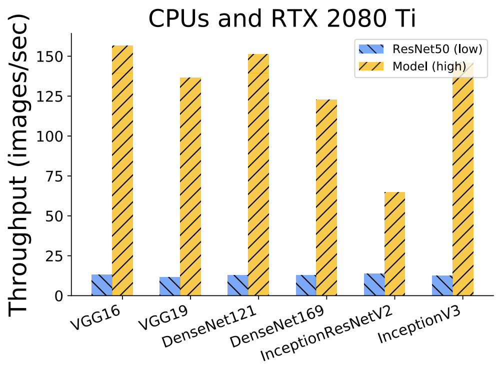
(a) Threaded TF: Co-train with ResNet50.



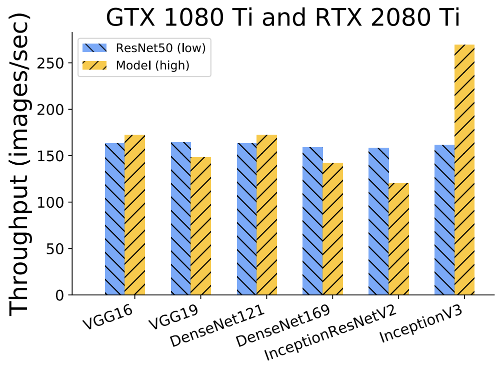
(b) Threaded TF: Co-train with VGG16.



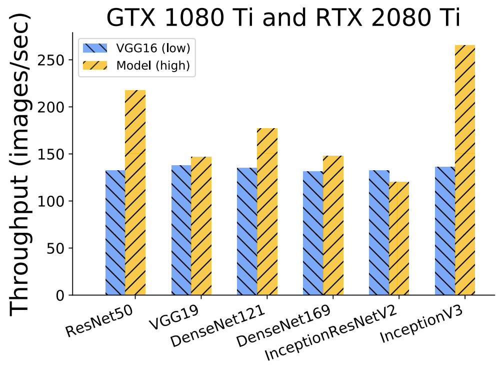
(c) MPS: Co-train with ResNet50.



(d) Co-train with ResNet50 (low).



(e) Co-train with ResNet50 (low).



(f) Co-train with VGG16 (low).

Figure 2-7: The throughput of two training jobs sharing a single GPU. The high priority job is shown in data series and the low priority background job is indicated in each figure. In (a)-(c), the bars in each group correspond to the performance of the two models, with the arrows showing the performance degrations compared to running in solo. (a) and (b) are under multi-threaded TF, (c) is under MPS, and (d)-(f) are under SwitchFlow.

sity. The key advantage of SwitchFlow is the isolation between training and inference jobs.

Table 2.1: The overhead of model state transfer.

Model Name	Stateful Variables (MiB)	Transfer time (ms) GPU to GPU (PCIe 3.0)
ResNet50	198.53	28.838
VGG16	1055.58	103.747
VGG19	1096.09	109.416
DenseNet121	64.83	39.823
DenseNet169	108.61	45.236
InceptionResNetV2	426.18	82.137
InceptionV3	182.00	31.613
MobileNetV2	27.25	17.505

Throughput

We are also interested in the performance of a preempted job and evaluated the throughput of two co-running training jobs. We considered a scenario in which a high priority training job needs to preempt a low priority job to run on a GPU. The GPU could be the only one on a machine or the faster one among multiple GPUs. In the vanilla TF that does not support GPU sharing, the low priority job has to be killed. Therefore, we used the multi-threaded TF as the baseline, under which the two models can freely share GPU. We also compared SwitchFlow with NVIDIA MPS on the V100, the most powerful GPU in our testbed. Figure 2-7 (a) and (b) show that resource contention on GPU led to significant slowdowns to both models. More seriously, allowing models to freely access GPU resources resulted in some model crashes due to OOM errors on both GPUs. Users need to carefully determine which models cannot be collocated. This is a tedious process since memory demands also depend on model input. Multi-threaded TF causes OOM errors when the aggregated, real-time memory demand of the two models at any point exceeds device memory. Worse, all models crash under NVIDIA MPS on the 1080 Ti and 2080 Ti GPUs because the two processes in MPS, each running a separate model, do not share GPU memory allocation. Thus, when the aggregated peak memory demand exceeds GPU

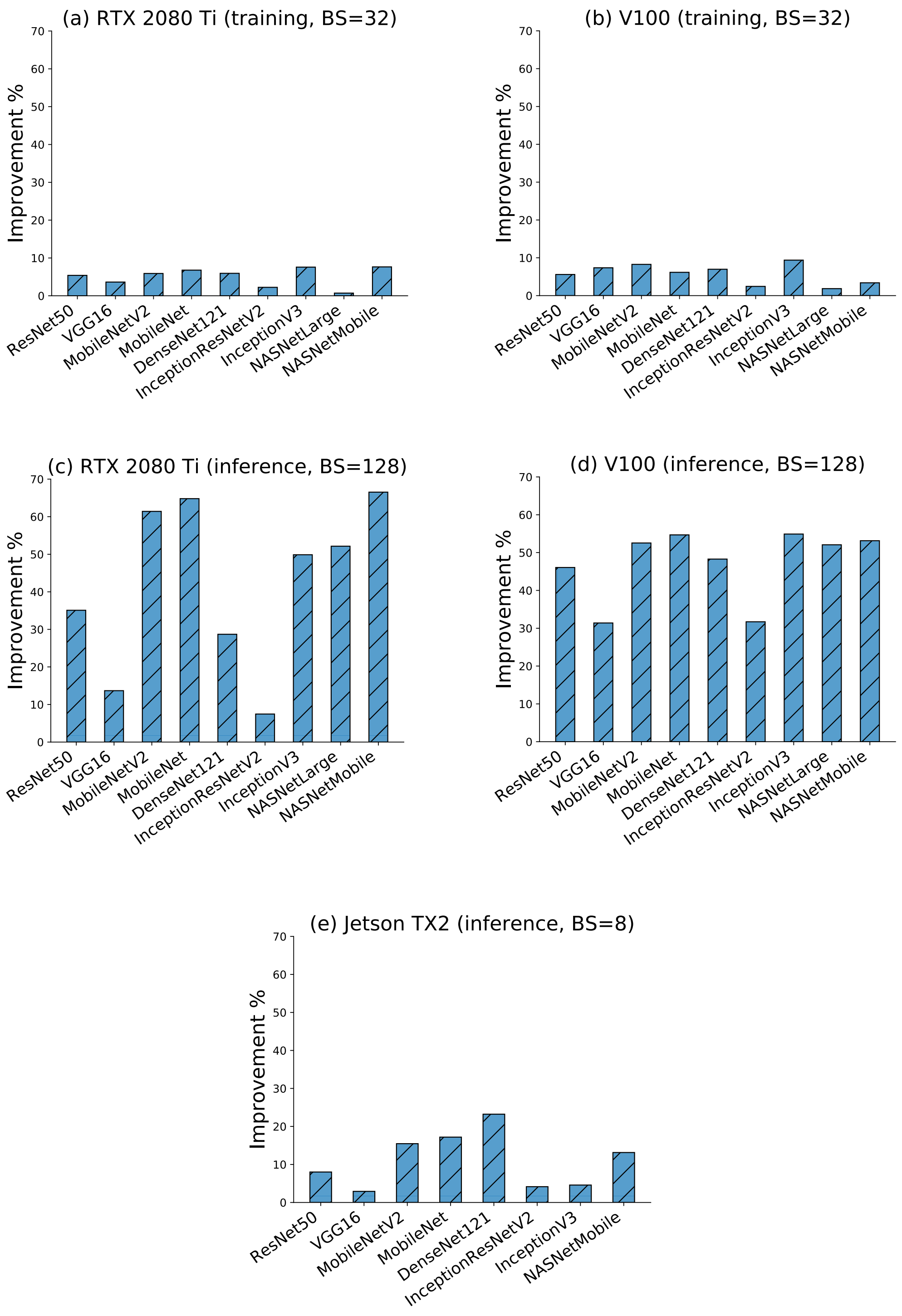


Figure 2-8: Performance improvement due to input reuse among multiple identical models against session-based time slicing.

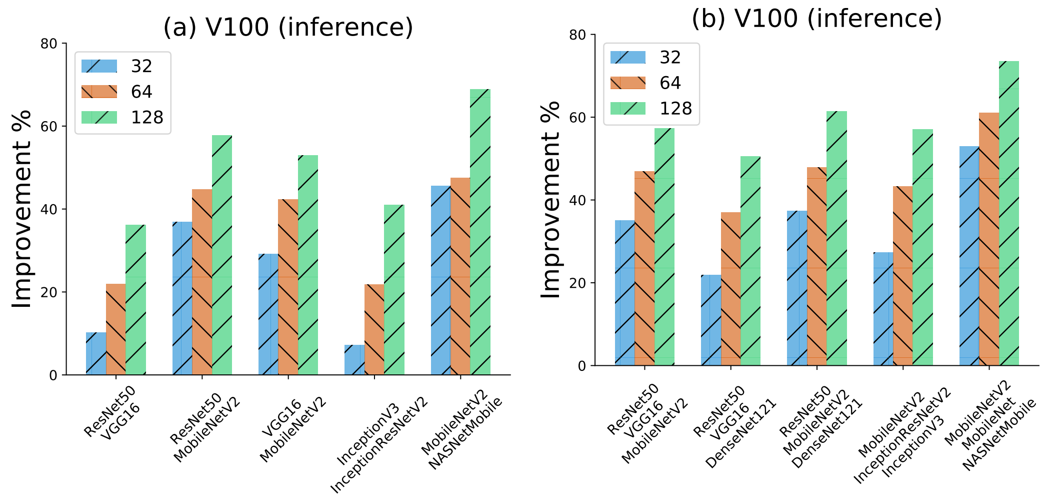


Figure 2-9: Performance improvement due to input reuse among different models.

capacity training crashes. Model co-training under MPS only can complete on V100 since it has triple device memory. Similar to multi-threaded TF, MPS also inflicted significant slowdowns to both models.

In contrast, SwitchFlow does not require user-side tuning and allows models to access full GPU capacity. Upon the arrival of a high priority model, the low priority one is preempted and migrated to a different device, whether be another slower GPU or CPU. We make the following observations in Figure 2-7 (d)-(f): 1) there was no crash. 2) in all cases, the high priority job achieved much higher throughput than that in multi-threaded TF. 3) The low priority job achieved acceptable throughput when migrated to a slower GPU but suffered drastic throughput drop when migrated to CPU since the DL operations were not designed to run on CPU. 4) The high priority job still experienced throughput loss compared to running in solo. While it ran a dedicate GPU, the low priority job occupied a few worker threads, which may delay task dispatching in the high priority job.

Preemption Overhead

Preemption in SwitchFlow involves aborting the execution of outstanding nodes (kernels), allocating space on a destination device, transferring model states (weights) to the destination, and freeing the memory of model states on the source device. Only waiting for the outstanding nodes of a preempted job to complete is on the critical path of a new job. Figure 2-3 shows kernel time ranging from a few tens of milliseconds. Therefore, the worst case preemption latency is approximately a few tens of milliseconds. The aborted operations are stochastic when preemption occurs, so the preemption latency is implicitly subject to the worst case operation.

Another source of overhead is the memory space needed to retain the model states of the preempted job until they are transferred to the destination device. Since the state transfer is asynchronous, the retained states occupy the GPU memory that could otherwise be used by a new job. Table 2.1 shows the amount of data need to be transferred as model states and the time required for GPU-to-GPU transfer via x16 PCIe 3.0. The largest model (VGG19) occupied about 10% of the GPU memory, e.g., 11GB device memory on GTX 1080 Ti and RTX 2080 Ti, and it takes at most 110ms before the states are transferred and memory is released.

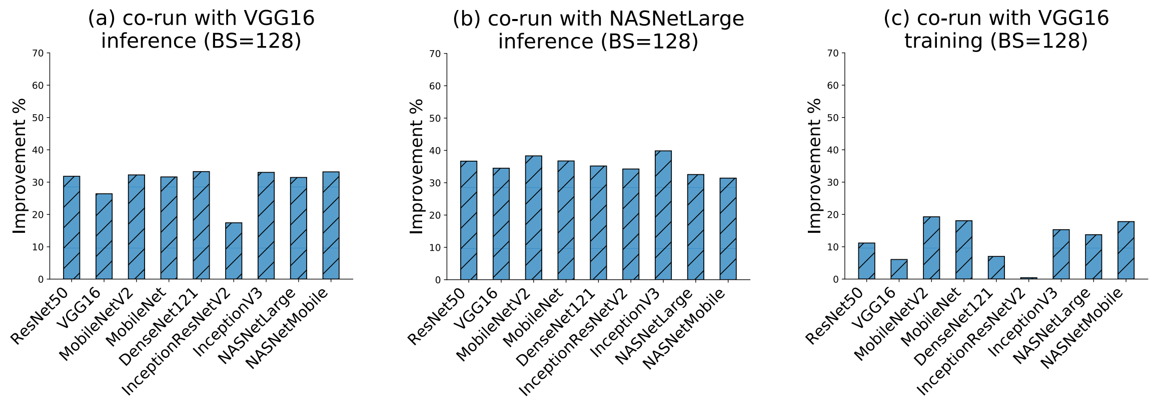


Figure 2-10: Performance improvement due to interleaving executors of independent models against session-based time slicing.

2.5.3 Sharing Inputs among Similar Models

Allowing models to simultaneously share GPU may cause OOM errors while dedicating heterogeneous devices to one model leads to low efficiency. In this section, we evaluate the efficacy of sharing the data preprocessing stage among models that take the same input batches and can be trained in a lockstep manner [104]. SwitchFlow is configured to alternate model executions before moving to the next input batch. The baseline is **session-based time slicing**, an approach adopted by Gandiva [130], wherein models have exclusive access to both CPU and GPU during one session run and each model is allowed one session run at a time. Note that there is no data reuse in the baseline. Performance is measured by the completion time of 200 iterations (training or inference) from each model after warmup.

In cases where multiple models have same preprocessing pipelines, to mitigate the upstream data preprocessing [82, 28] time, batched input data are reused between different models. As downstream GPU keep consuming input data without involving repeated data preprocessing, training or inference workloads which are bottlenecks at CPU side can achieve speedup. Initially, the master model carries out data preprocessing and data augmentation. Next, the processed input are cached for the subsidiary models to exploit again in the following session runs.

Figure 2-8 shows normalized performance improvement due to SwitchFlow against the baseline on three GPUs. In this evaluation, we co-ran two identical models as their sessions are guaranteed to have the same length so that the maximum gain of input reuse can be determined for each specific model. Figure 2-8 (a) and (b) suggest that there was marginal performance gain due to input reuse for training jobs. Since each iteration of training lasts hundreds of milliseconds on GPU, the existing mechanisms in TF, such as input prefetching and parallel data preprocessing, can effectively overlap GPU and CPU time, leaving little room for further improvements. In contrast, Figure 2-8 (c) and (d) show significant improvements when two inference jobs were collocated. Input reuse saved as much as 65% compared to session-based time slicing. An interesting observation is that faster GPU (V100) led to higher gain in complex models (e.g., ResNet50, VGG16, and InceptionResNetV2) but lower gain in lightweight models (e.g., MobileNet and NASNetMobile). Jetson TX2 has limited shared memory between CPU and GPU and thus is not intended for training. Figure 2-8 (e) shows lower gain for inference on TX2 as the embedded GPU is much slower. Again, the higher gains were from lightweight models, which require less GPU computation.

Next, we evaluate input sharing among different models. Although models have different internal structures, they are CNN models for image classification. Thus, they can share the preprocessing stage. Figure 2-9 shows the results with different batch sizes and a varying number of collocated models. The findings are 1) larger batch sizes led to higher improvements, indicating CPU increasingly became the bottleneck as more images included in a batch. 2) Co-running more models had diminishing gains, especially among complex models. According to the results, it is recommended that no more than three models should co-run on a single GPU.

2.5.4 Interleaving Independent Models

In this section, we relax the requirement of sharing the input and evaluate how much SwitchFlow improves executor scheduling among independent models. SwitchFlow alternates GPU executors from different models but allows CPU executors to freely run. Figure 2-10 shows performance improvements in three scenarios: (a) inference jobs sharing with inference of a heavy-weight model (VGG16), (b) sharing with inference of a lightweight model (NASNetLarge), and (c) sharing with training of a heavy-weight model (VGG16). The GPU used was V100. The figure shows that SwitchFlow's performance gain compared to the baseline was much lower than that in the input reuse tests. This is expected since scheduling may not perfectly overlap GPU and CPU processing while sharing inputs entirely bypasses the CPU stage. Still, SwitchFlow was able to consistently achieve 30% among inference jobs, regardless of the model type. When co-ran with training, the gain diminished but for lightweight models (e.g., MobileNetV2) the gain was up to 20%.

2.6 Related work

Various approaches to share GPUs in a multitasking environment are proposed to meet a number of objectives, such as responsiveness, throughput, resource utilization, isolation.

Temporal and spatial GPU multitasking. Existing studies in GPU multitasking include: (1) time-sliced scheduling [90]; (2) spatial partitioning scheduling [7, 92]; (3) space-time scheduling [70, 57, 84]. The proposed strategies can be categorized into different scheduling granularity: context level [124], kernel level [57, 121], thread block level [24], SM level [133, 127], and graph nodes level for DL workloads [52]. Time-sliced scheduling controls the state transitions, priority to ensure responsiveness and fairness. Interrupt request triggers context switch between a serial of applications. Spatial partitioning scheduling relies on data slicing, kernel slicing and fusion to split data and kernel into a number of smaller chunks so that they can co-schedule sub-kernels to different CUDA streams or SMs.

These work focus on low-level management of GPU kernels and memory copy. SwitchFlow takes both low-level kernel launching constrains and DL DAG computation graph characteristics into consideration to enforce time-slicing a GPU exclusively. Thus, high GPU utilization can be achieved without interference.

DL workloads scheduling, preemption and migration. ByteScheduler [95] is a generic priority-based scheduler for DNN distributed training. Gandiva [130] is a cluster scheduling framework to improve latency and efficiency of training DNN models by time-slicing GPUs. Olympian [52] proposes a scheduling policy to enable fair sharing multiple DNNs in TF-Serving [88]. Pretzel [67] applies multi-model optimizations for ML.Net [9] prediction serving systems. Previous work consider either training or inference phases, SwitchFlow instead focuses on both workloads to maximize GPU utilization and throughput, and to minimize latency. While PipeSwitch [14] enables fast model switching to allow multiple DL models to share a single GPU, it relies on dynamic graph execution for layer-to-layer model transmission and execution. In contrast, SwitchFlow focuses on DL multitasking on static graphs, which are more efficient but challenging to switch. AntMan [131] proposes elastic memory management, which can potentially help in DL multitasking and is orthogonal to SwitchFlow. However, it requires unified GPU memory and may incur high overhead.

Olympian [52] interleaves with graph nodes but do not preempt ongoing graph.

In Gandiva [130], preemption and migration is extended by the Tensorflow already supported checkpoint APIs, which may incur considerable overhead compared with SwitchFlow by saving and restoring several hundreds of MiB or few Gib checkpoint [130] that cannot be tolerated for inference jobs. Our design does not preempt an issued GPU kernel since it can be expensive [91, 128, 115]. We transfer stateful variables to another device without involving checkpoint. Also, DL systems consists of data preprocessing pipelines for both training and inference [28]. SwitchFlow leverages overlapping data preprocessing and kernel execution to maximize throughput.

2.7 Conclusion

This chapter presents SwitchFlow, a scheduling framework for DL multitasking. Spatial and temporal multitasking are either ineffective or inefficient in DL frameworks that employ computation graphs. We demonstrated that by carefully controlling the scheduling of GPU executors, one can simultaneously achieve high pipeline efficiency and OOM-free execution. We evaluated SwitchFlow with representative DNN models. The results show that SwitchFlow achieved significant performance improvements on both inference latency and training throughput compared to TensorFlow.

Chapter 3

Atom: Asynchronous Training of Massive Models for Deep Learning in a Decentralized Environment

With the emergence of the Transformer architecture, natural language processing (NLP) models have seen significant growth and achieved outstanding success across a wide range of challenging NLP tasks. However, the lack of specialized hardware resources such as large GPU memory and high-speed interconnects presents a challenge for training large-scale models. As a result, pre-training and fine-tuning large language models (LLMs) can be difficult for the average user to experiment with new ideas. In this chapter, we propose Atom , an elastic, fault-tolerant distributed training framework that supports asynchronous training of massive-scale models in a decentralized environment using more affordable hardware such as consumer-class GPUs and Ethernet. Unlike existing model partitioning approaches that train sub-models on distributed GPUs, Atom aims to accommodate an entire LLM on a single host through model swapping and train a large number of copies in parallel to achieve high training throughput. Atom uses static analysis to determine an optimal model partitioning scheme and schedule that seamlessly overlaps model execution with swapping. Atom offers several advantages: 1) it eliminates the single point of failure in pipeline parallelism approaches; 2) its independent architecture proves to be more performant and scalable than tightly-coupled pipeline parallelism with low-speed networks. Our experiments with various configurations of the GPT-3 model show that Atom improves training performance by as much as 20× for poor network connections compared to state-of-the-art decentralized pipeline parallelism approaches.

3.1 Introduction

Deep Neural Networks (DNNs) have advanced significantly in recent years with the use of deeper layers and larger model capacity, leading to improved performance in a variety of machine learning tasks such as computer vision [51], natural language understanding [21], structural biology, and drug discovery [62]. The introduction of Transformer models [120] has further enhanced the capabilities of DNNs, enabling the development of large language models (LLMs) that can be applied to a wide range of applications, such as the generation of programming code with OpenAI Codex [25] and GitHub Copilot [41].

Recent advances in natural language processing (NLP) have led to the development of large models such as BERT-Large [35], GPT-2 [97], Megatron-LM [112], T5 [98], and GPT-3 [21], with model sizes ranging from 0.3 billion to 175 billion parameters. These models require significant resources to train from scratch, such as the use of thousands of accelerators and large text datasets. For example, GPT-3 175B, which has 175 billion parameters, was trained using 45 TB of text data and required over one month to complete training. This is a significant increase in model size compared to image recognition models such as ResNet-50 [51] which has 25.6 million parameters, outpacing the development of computational, networking, and storage hardware.

While training large language models from scratch can be a challenging task for individual users and small- or medium-sized institutions and companies, there is an increasing effort to make pre-trained LLMs publicly available [89, 73, 135]. This allows users to experiment with pre-trained models and adapt them to their own datasets. However, LLMs are too large to fit on a single GPU, so researchers have been exploring ways to split the model and perform training in a distributed manner. This includes techniques such as model parallelism [110, 68, 132, 11] and pipeline parallelism [54, 78, 38] which divide the model into sub-models and train them on multiple GPUs. These approaches enable the use of LLMs by leveraging the device memory on distributed GPUs.

Despite the efforts to make pre-trained LLMs publicly available and the development of techniques to split and train them in a distributed manner, there are still challenges for average users to adopt these approaches. First, access to a large number of GPUs to accommodate LLMs is still limited to a few large institutions. Second, tight coupling between sub-model training on different GPUs is a major issue, requiring not only high-speed interconnects for transmitting model output and intermediate data, but also creating a single point of failure. Third, in cases where nodes can dynamically join and leave, a model needs to be re-partitioned before training can be resumed, and this can lead to a restart of training. Fourth, if the network bandwidth is constrained, it can limit the training performance.

In this chapter, we propose a solution for training large language models (LLMs) on a small number of commodity GPUs in a decentralized network environment with low-bandwidth and unreliable connections. Our key insight is that, although LLMs are extremely large in size, individual operators/layers, the building blocks of a computation graph representing the model, can be easily fit in a single commodity GPU. This opens up opportunities to execute the computation graph layer by layer through device-to-host memory swapping. Theoretically, even the GPT-3 175B model can fit in a single server given sufficient host memory. We explore an alternative approach to distributed training using loosely coupled GPUs, each training a complete LLM through memory swapping, to process massive mini-batches in parallel.

To this end, we present Atom , a distributed training approach that supports asynchronous training of massive-scale models in a decentralized environment. As shown in Figure 3-1, Atom takes a computation graph of an LLM[1] as input and par-

[1]In this chapter, we focus on using GPT as the large language model (LLM) due to its widespread usage and popularity in the field of natural language processing.

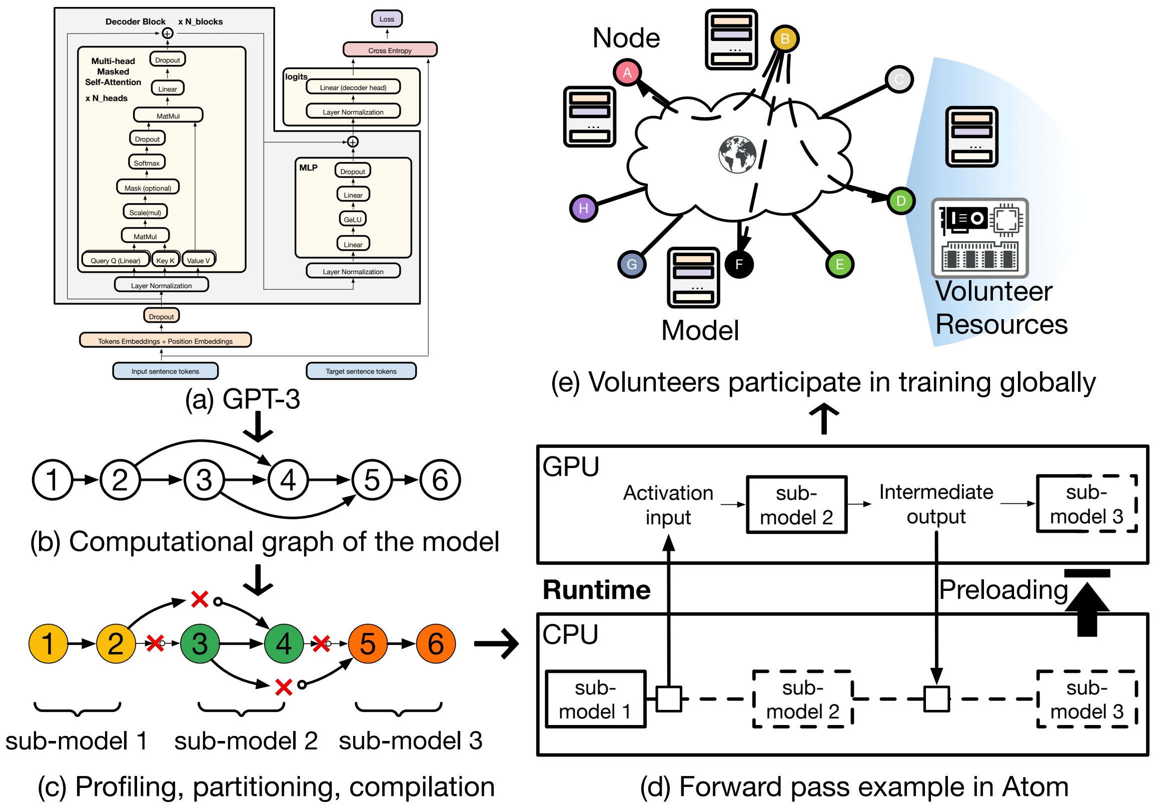


Figure 3-1: The figure illustrates the process and functionality of the Atom system for training large language models. The system is composed of several components: (a) the user model, (b) the automated partitioning of the computation graph, (c) the compilation of sub-models code, (d) the Atom runtime for swapping sub-models between host CPU and GPU, and (e) the ability for independent participants to join and leave the training process without disrupting it. The overall goal of the system is to efficiently and effectively train large language models while addressing the challenges of limited resources and flexibility.

titions the graph into sub-models based on a layer-by-layer profiling. Atom automatically generates an optimal schedule of the sub-models to streamline model execution and loading. Atom addresses the following challenges. *First*, memory swapping has been long criticized for low performance and high overhead. Atom avoids swapping overhead by asynchronously scheduling swapping and prefetching upcoming layers. To avoid GPU idleness, we also overlap the execution time of sub-models and the loading time of an upcoming sub-model. *Second,* searching an optimal model partitioning is difficult and time-consuming. Atom addresses this issue via a heuristic exhaustive search algorithm that integrates domain knowledge about the GPT-3 model. For elasticity and fault tolerance, Atom uses a DHT (distributed hash table) and a global batch size to coordinate the training on independent participants, which allows nodes to join and leave.

We implement Atom using Pytorch and Hivemind [118]. Evaluation results using GPT-3 model configurations and various network conditions show that Atom consistently outperforms Petals [18] by using the schedule policy of GPipe [54] and PipeDream [78] by a large margin. Experiments with three types of commodity GPUs demonstrate that the loosely coupled distributed architecture of Atom is more performant and scalable than the tightly-coupled pipeline architecture, especially when the interconnect is slow.

3.2 Background and Related Work

In deep learning frameworks, such as TensorFlow [6] and PyTorch [93], model training is usually represented as a computation graph wherein nodes are operators, such as matrix multiplication, and edges are dataflows connecting individual operators or a group of operators (called a layer). Each training iteration comprises of a forward propagation and backward propagation. In forward propagation, a mini-batch of training examples from a dataset is fed into the model from the first layer to the last layer to compute a loss value. In backward propagation, the gradient for each model parameter is derived from the loss value from the last layer to the first layer reversely. In the optimization step (e.g., using the stochastic gradient descent (SGD) algorithm), each model parameter is updated by adding a scaled gradient controlled by a learning rate and other hyper-parameters.

3.2.1 GPT-3 and the Transformer Structure

GPT-3 [89, 21] is an auto-regressive language model with 175 billion parameters (a.k.a. GPT-3 175B) that consumes 700 GB memory with 32-bit single precision floating point (`f32`) parameters (4 bytes). GPT-3's training set contains 499 billion tokens from datasets Common Crawl, Wikipedia, and WebText2, etc. The input of GPT-3 is a sentence of tokens (words) $(x_0, x_1, \ldots, x_{L-1})$, and the output x_L coming at the end of the sentence is a token (word) predicted with the highest probability to complete the sentence. The GPT-3 model is a stack of 96 decoder blocks based on the attention mechanism [13], a key module in GPT-3 drawing global dependencies of hidden representations between different positions of the input and output. Decoder is from the Transformer [120] encoder-decoder architecture. Each decoder block has a multi-head masked self-attention module and a fully connected feed-forward layer or multilayer perception (MLP). The loss value between the input sentence and target sentence tokens is computed after decoder head.

Training a GPT-3 model requires a daunting amount of memory. Not only GPT-3 model parameters (700 GB in GPT-3 175B), including those in token embeddings, positional encodings, transformers, and fully-connected layers, need to be resident in memory, training also generates a large amount of intermediate data, mainly in optimizer states. In general, the optimizer states are about $3\times$ of the size of model parameters, leading to a total memory size of more than 2.8 TB. Apparently, GPT-3 cannot fit in a single GPU.

Table 3.1: Comparison between different large model training schemes.

Training Schemes	Support LLMs	Host memory extension	Slow network prone	Elastic	Fault tolerance
Pipeline Parallelism approaches: GPipe [54], PipeDream [78], FTPipe [38], Megatron-LM [112], Terapipe [69]	✓	✗	✗	✗	✗
DeepSpeed [116], ZeRO-Offload [117], FSDP [74], SwapAdvisor [53], HUVM [30], vDNN [103], Capuchin [94]	✓	✓	✗	✗	✗
Alpa [137], GSPMD [132], GShard [68], TF-Mesh [110]	✓	✗	✗	✗	✗
Ray [77]	✗	✗	✗	✓	✓
Hivemind [106], DeDLOC [36]	✗	✗	✓	✓	✓
Swarm [105], Bloom [108]	✓	✗	✓	✗	✓
Atom (Our work)	✓	✓	✓	✓	✓

3.2.2 Volunteer Computing

Volunteer computing (VC) [107, 36, 105] is a distributed training paradigm to harvest geographically idle computing power and storage resources without advanced computing and networking devices. The academic institutes [119, 10] and communities [37, 55] seek to democratize the access to large language models via collaborative training on voluntary resources. Hivemind [106, 19, 36] trains a Mixture of Experts (MoE) [111, 68] model in an asynchronous manner by volunteer device over a decentralized network. Distributed Deep Learning in Open Collaborations (DeDLOC) [36] demonstrates the feasibility of collaborative training using model ALBERT [63]. Nevertheless, these VC frameworks are only able to train small models that can fit entirely in any volunteer device. Petal [18] and Swarm [105] seek to extend pipeline parallelism to a large number of heterogeneous volunteer servers. While they focus on fault tolerance in distributed training, it does not adequately address the low network bandwidth and high latency in a decentralized network, thereby unable to efficiently perform LLM training.

Table 3.1 summarizes and compares Atom with various existing distributed training schemes, and highlights their main pros and cons when adopted in decentralized LLM training.

3.2.3 Distributed Training

Distributed training distributes training workloads across multiple GPUs to either reduce training time or accommodate large models. *Data parallelism* [109, 60, 77] splits up the training data set on multiple GPUs and processes them in parallel. Each GPU trains a copy of the model on a different batch of the training data and communicates the gradients to keep model parameters in sync. Since each GPU maintains a complete model, data parallelism does not address large model size.

In contrast, *model parallelism* partitions a model into sub-models and places them on separate GPUs. Sub-models are trained on different GPUs using the same training data following the topological order in the model's computation graph. Model parallelism reduces the memory requirements for individual GPUs, allowing much larger models to be trained across GPUs as long as the aggregated GPU device memory can accommodate the complete model size and sub-models can fit in each GPU. However, model parallelism suffers two drawbacks. First, due to the dependencies between sub-models, only one GPU is active at a time executing a single sub-model, causing low GPU utilization. Second, activations, i.e., the output of a layer, need to be communicated to a downstream sub-model, which requires significant bandwidth between GPUs. In cases where a single layer cannot fit in a GPU, intra-layer partitioning [110, 68, 132, 11] is needed and incurs even more inter-GPU communication [38].

To address low GPU utilization in model parallelism, *pipeline parallelism* similarly partitions a model into sub-models but allows them to be executed simultaneously on different GPUs with different training mini-batches. There are various pipeline parallelism designs and they differ in how model parameters are updated by distributed workers (GPUs). *Synchronous training* is an approach where all workers train the same global model in lockstep on every mini-batch while in *asynchronous training*, each worker trains its own model on multiple mini-batches and periodically updates the shared global model through all-to-all communication. In general, synchronous training converges faster than asynchronous training but the training throughput, i.e., the number of mini-batches processed per unit time, and GPU utilization are lower.

Pipeline parallelism. Figure 3-2 depicts two representative pipeline parallelism schemes, GPipe [54] and PipeDream [78], overlap the processing of multiple mini-batches (numbered 1 through 7). The model is divided into four sub-models and distributed on four workers. As shown in Figure 3-2(a), GPipe adopts synchronous pipeline parallelism and accumulates the gradients calculated in the forward propagation for four mini-batches before the backward propagation can start. The backward pass reversely updates model parameters from mini-batch 4 through 1. Due to synchronous training, there is significant idleness (denoted as white spaces) on the workers between the forward and backward propagation. In contrast, PipeDream

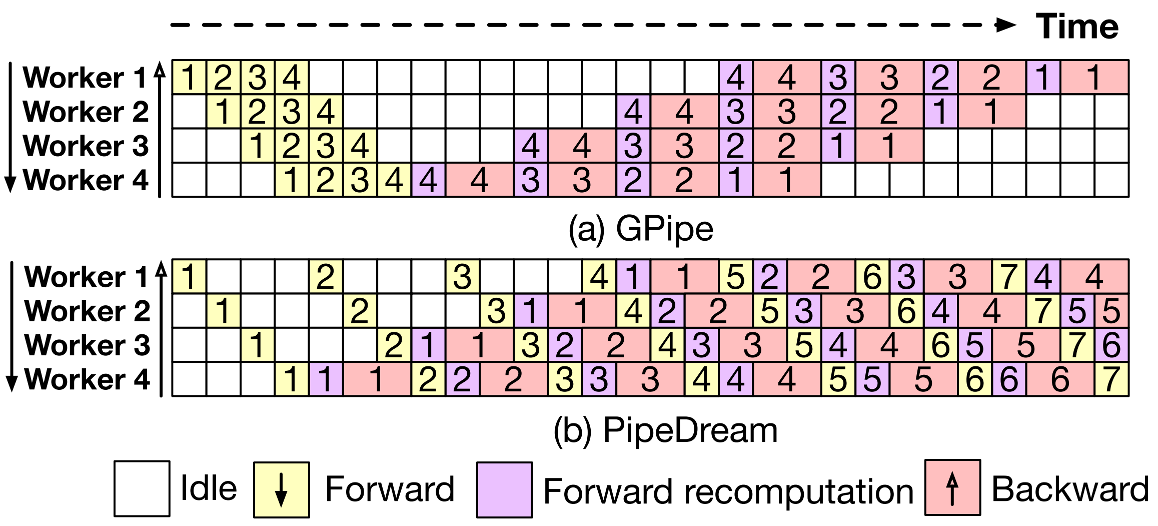


Figure 3-2: GPipe and PipeDream.

employs asynchronous training and allows workers to process different mini-batches while model updates due to other mini-batches are ongoing. This leads to high GPU utilization (as shown in Figure 3-2 (b)) though at a cost of slower convergence. [38] is an asynchronous pipeline parallelism approach similar to PipeDream that relaxes the topology constraints to allow non-adjacent layers to be placed on the same GPU. While FTPipe embraces higher GPU utilization, it relies even more heavily on a fast interconnect.

GPU memory saving. While distributed training partitions models among GPUs, sub-models could still be too large to fit in a single GPU. Various GPU memory saving techniques are proposed to further reduce the footprint of sub-models in GPU device memory. ZeRO (Zero Redundancy Optimizer) is a family of memory optimization approaches [99, 117, 100] for deep learning under the DeepSpeed [116] framework. ZeRO [99] and FSDP [74] both split model states, including model parameters, gradients, and optimizer states, and distributes them on multiple GPUs. Model states are *fetched on-demand when they are needed to update a particular sub-model at the cost of high bandwidth intra-node communication.* ZeRO-Offload [117] moves the intermediate data generated during training, i.e., gradients and optimizer states, from device memory to host memory to alleviate GPU memory pressure. ZeRO-Infinity [100] further exploits CPU, GPU, and NVMe memory to allow even larger model training.

Another common technique employed in GPipe and PipeDream is forward recomputation during the backward propagation (shown as the duplicate mini-batch processing in purple color in Figure 3-2). Instead of storing activation outputs and intermediate data for each sub-model, which are usually $3\times$ of the size of sub-model parameters, they are recomputed in the backward propagation.

Limitations. *First*, the existing distributed training schemes require high-speed interconnect, such as NVLink [86], Infiniband, and 40Gbps Ethernet, to effectively construct a training pipeline that overlaps sub-model computation with the transmission model states between sub-models (GPUs). However, advanced network devices are as scarce and expensive as high-end accelerators, rarely accessible to average users. *Second*, there is a lack of fault tolerance in the design of distributed training and failures at any workers will stall the entire training pipeline. *Third*, a model must be statically partitioned offline before execution. Any changes to the number of workers or the network condition requires the model to be re-partitioned, lacking flexibility and elasticity.

3.3 Atom

In this section, we shed light on how Atom addresses the drawbacks of distributed training schemes designed for high-performance data centers and volunteer computing to enable elastic, fault tolerant yet efficient training of LLMs over commodity networks. We first characterize the memory usage and computation requirements for each layer in model GPT-3 and elaborate on how the insights obtained in the profiling guides Atom design. Without loss of generality, we discuss Atom's design in training model GPT-3 but note that Atom can be extended to other complex deep learning models.

3.3.1 Design Overview

Atom centers on a key design different from the existing distributed learning schemes. Instead of distributing partitioned sub-models on different GPUs, Atom maintains a complete model on a single server in host memory and only loads a sub-model on GPU device memory when needed via memory swapping. Unlike the existing schemes that the aggregated device memory on distributed GPUs to accommodate LLMs but at a cost of communication overhead, Atom leverages host memory, which is more accessible for average users than a large number of accelerators and high-speed networks, to host LLMs. Models trained locally at each GPU are synchronized through an allreduce communication when participating GPUs reach a predefined global batch size.

Atom does not have a single point of failure and is able to make training progress in the presence of worker failures, joining, and departure. In what follows, we elaborate on the profiling results of model GPT-3 that motivated Atom design (§ 3.3.2), the design of sub-model swapping that achieves high GPU utilization (§ 3.3.3), and automated model partitioning and code generation (§ 3.3.4).

3.3.2 Characterization of GPT-3

We perform a comprehensive profiling of GPT-3 to measure the memory footprint, the execution time, and the memory swapping time of its key operators/layers and their associated data during training. We study the feasibility of partitioning GPT-3 to fit in a single GPU and streamlining sub-model execution and switching. The full version of GPT-3 175B includes a chain of 96 identical decoders and requires approximately 2.8 TB memory. To fit GPT-3 in our server, which is configured with 384 GB host memory, we trimmed GPT-3 175B down to including two decoders. Note that the trimmed GPT-3 model has identical structure in model execution, including the profiling of one of the 96 identical decoders, but differs in the accuracy of the trained model. We constructed the GPT-3 computation graph[2] through PyTorch module `torch.nn.Module` and enabled tracing. Layers and operators were executed according to the graph's topological order. All results were the average of 10 runs. The experiments were conducted on an NVIDIA Tesla V100 GPU with 32 GB device memory running on a PCI-e 3.0 bus.

[2] https://github.com/karpathy/minGPT

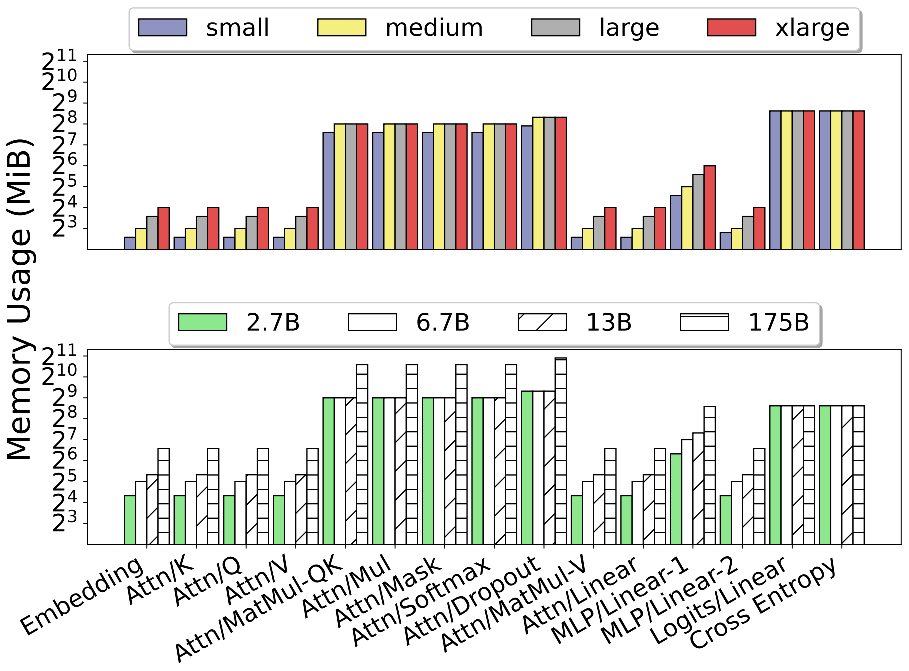


Figure 3-3: GPT-3 memory usage in forward propagation.

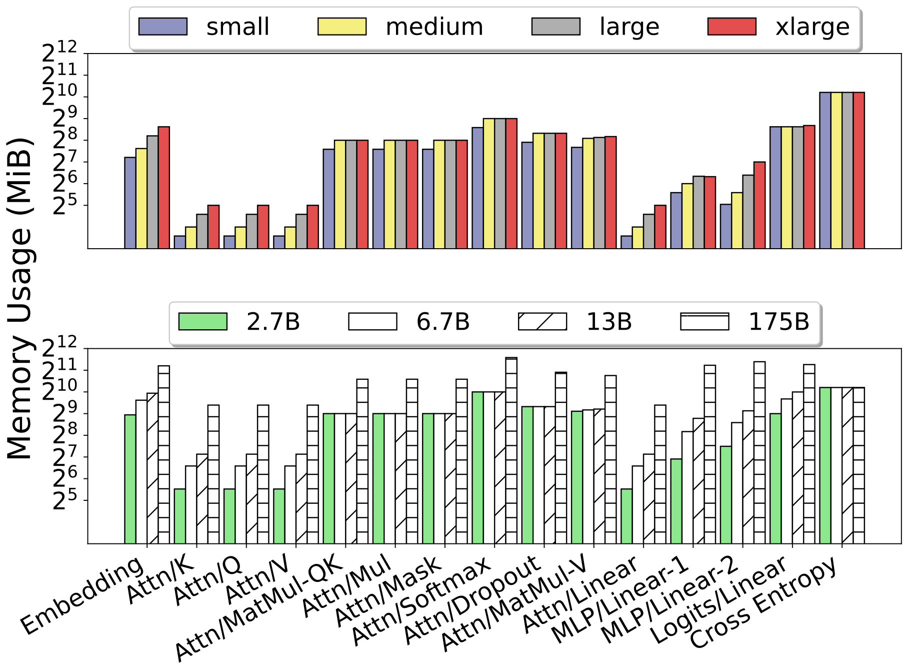


Figure 3-4: GPT-3 memory usage in backward propagation.

Memory usage breakdown

Figure 3-3 and Figure 3-4 show the memory footprints of individual layers in GPT-3 in forward propagation and backward propagation, respectively. We show eight variants of GPT-3 ranging from 125 million (small) to 175 billion (175B) parameters. Memory usage was measured by `reset_peak`, `_memory_stats`, and `max_memory_allocated` from the PyTorch library `torch.cuda`. The measured memory consumption of each layer includes activation inputs, intermediate outputs, model parameters, and memory for temporary workspace at runtime.

Inspired by ZeRO memory optimizations, we configured GPT-3 to offload optimizer states to the host memory and performed model updates on CPU. Thus, the memory footprints shown in Figure 3-3 and Figure 3-4 exclude the optimizer states. As shown in the figures, the peak memory usage in GPT-3 175B are the *dropout* layer (1,920 MB) in forward propagation and the *softmax* layer (3,072 MB) in backward propagation. The results suggest that even the most memory-intensive layer in the GPT-3 175B model can fit in an entry-level GPU, e.g., NVIDIA GTX 1080 with 8 GB device memory, without a need to further perform intra-layer partitioning [59, 68, 38]. Another observation is that if the optimizer states, which includes a copy of the model parameter (4 bytes), the momentum (4 bytes), and the variance (4 bytes) for each 4-byte `f32` model parameter, were not offloaded to host memory, the total size of a layer (4× of the peak memory size shown in Figure 3-3 and Figure 3-4) may exceed GPU capacity. This highlights the significance of optimizer state offloading. Switching the model to SSD or NVMe as a storage solution for GPT-3 175B is a topic of future research since GPT-3 175B requires a substantial amount of memory, more than 2 terabytes, which exceeds the memory capacity of most commodity personal computers.

Model partitioning may include multiple layers in a sub-model, which is expressed as a sub-graph of the computation graph and need to be independently compiled and optimized by the runtime. We further verified that the memory usage of a sub-model containing multiple layers is less than or equal to the sum of the memory usage of each individual layer. In summary, there always exists a model partitioning, in the worst case a layer-by-layer partitioning, that allows GPT-3 175B to execute on a consumer-class GPU.

Memory swapping vs. network transmission

As we have proved the feasibility of fitting a GPT-3 model on a single GPU, we next study the efficiency of executing LLMs on a single GPU compared with that on multiple GPUs over a network. For fair comparison, we ensure the number of and type of GPUs used for training remain the same. Atom executes complete copies of models on individual GPUs via sub-model swapping and uses allreduce communication to synchronize the models while pipeline parallelism approaches split the model on multiple GPUs and rely on network transmissions for sending activation outputs between sub-models. In this section, we compare the cost of sub-model swapping with that of sub-model activation via network transmission. A more comprehensive comparison considering the allreduce communication can be found in § 3.5.

Table 3.2: GPT-3 configurations to measure the cost of network transmissions in model partitioning.

GPT-3 Model	L	d_{model}	n_{heads}	activation payload (MiB)
Small (125M)	2048	768	12	6
Medium (350M)	2048	1024	16	8
Large (760M)	2048	1536	16	12
XL (1.3B)	2048	2048	24	16
2.7B	2048	2560	32	20
6.7B	2048	4096	32	32
13B	2048	5120	40	40
175B	2048	12288	96	96

To quantify the cost to transmit activation inputs and intermediate output between sub-models over network, we set up distributed training in Pytorch on two servers using asynchronous gRPC [45] APIs. The interconnect was 10 Gbps Ethernet and we used bandwidth throttling to evaluate transmission rates at 400 Mbps, 800 Mbps, and full-speed 10 Gbps. The 400 Mbps rate represents a typical wide-area

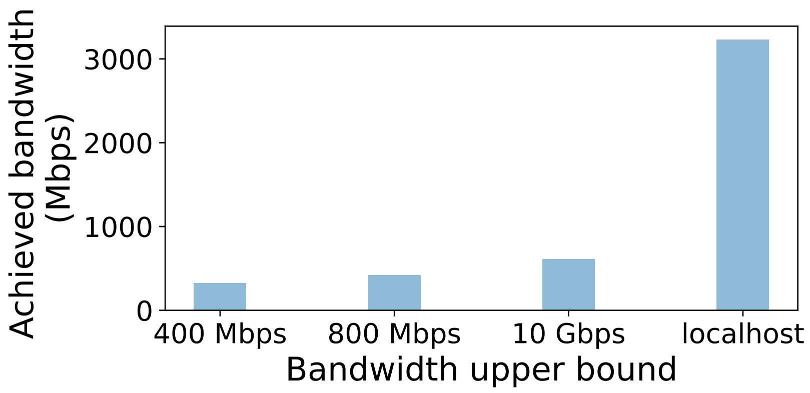


Figure 3-5: Achievable bandwidth in distributed training.

bandwidth in decentralized data analytics [123]. We also used the loopback network device (denoted as *localhost*) to derive a upper bound on the achievable bandwidth. We optimally partitioned GPT-3 models at the boundary of the transformer block (decoder) since it has the least activation output. This serves the lower bond on the amount of network transmission in any model partitioning. Table 3.2 shows the configuration of GPT-3 and the resulted activation payload that needs to be transmitted over network. The shape of the activation tensor is $[batch\_size, sentence\_length, d\_model]$, where batch size is set to 1, the input sequence length L is 2048, and the embedding dimension is d_model.

Figure 3-5 shows the achieved bandwidth by transmitting activation payloads between sub-models under various network conditions. Note that the actual bandwidth in distributed training is much lower than the line speed of the network. For the 10 Gbps network, activation transmissions are bottlenecked at 610 Mbps, which is also reported by [17][3]. Although gRPC is a widely adopted communication mechanism to implement distributed training, it requires activation payloads to be transferred from GPU to CPU for serialization before they are sent to a downstream sub-model. Figure 3-6 reports the round-trip transmission time, from GPU to CPU and to the GPU for the next sub-model, for activation payloads in various GPT-3 models.

In comparison, we measured the time to load key layers of the GPT-3 model from host memory to GPU device memory via memory swapping. Similar to activation

[3]a round-trip to invoke the remote sub-model functions

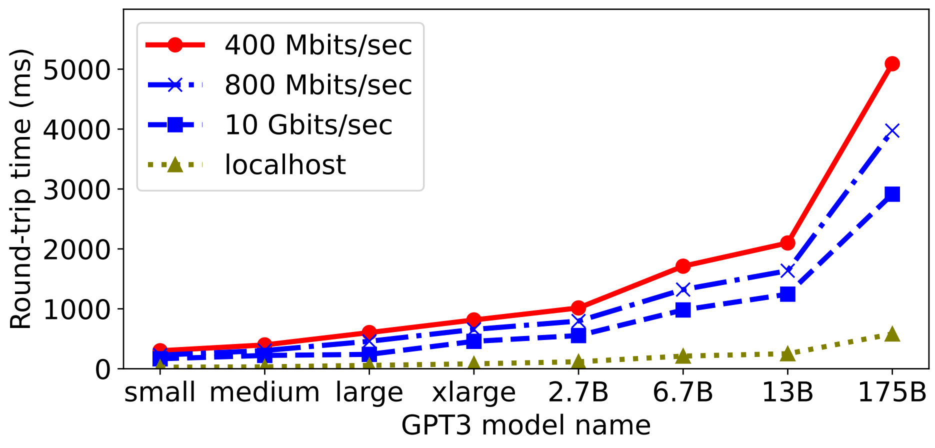


Figure 3-6: Transmission time of activation payload using gRPC in various GPT-3 models.

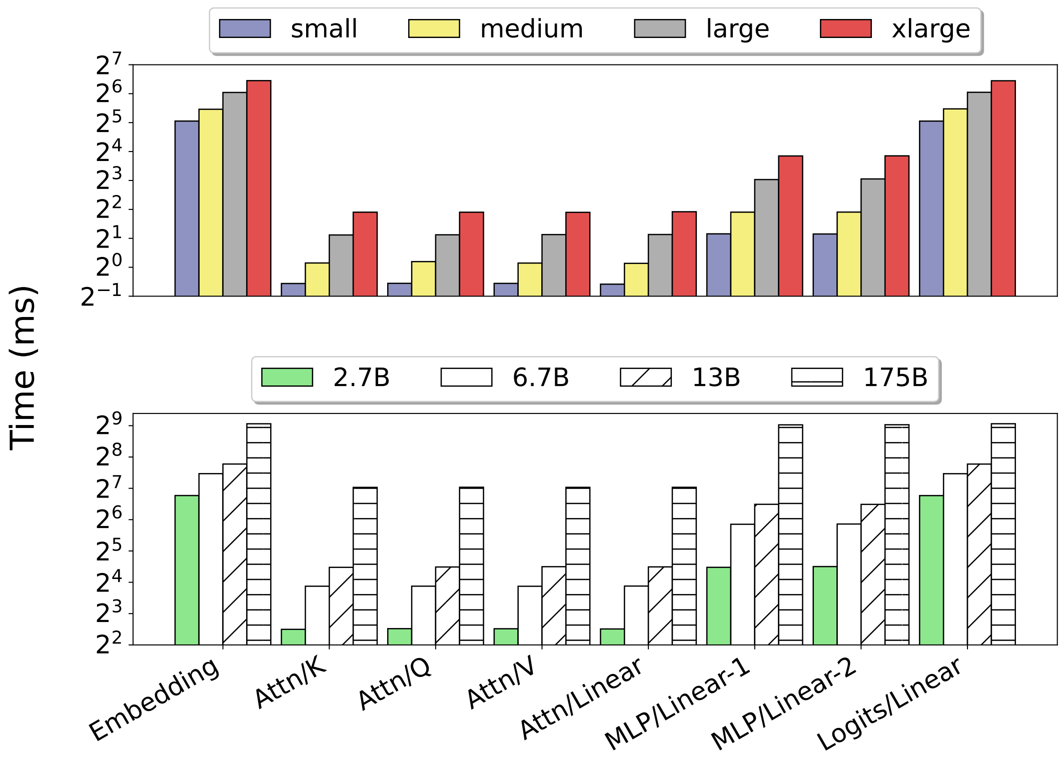


Figure 3-7: Layer loading time in various GPT-3 models.

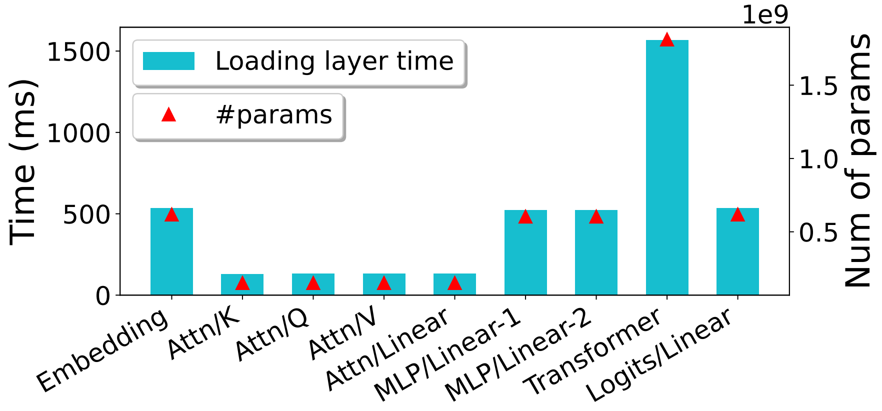


Figure 3-8: Relationship between loading time and layer size.

transmission time in model and pipeline parallelism, the layer loading time is on the critical path of model execution and significantly affects training performance and GPU utilization. Figure 3-7 plots the layer loading time in eight GPT-3 models. It suggests that the loading time of any layers, even for the largest GPT-3 175B model, is an order of magnitude lower than the reported activation transmission time in Figure 3-6. Note that the activation transmission time is due to an optimal partitioning at the transformer block, other partitioning schemes will incur even higher transmission overhead. While the size of the activation tensor to be transmitted depends on the type of activation output and how model partitioning is done, the layer loading time is more predictable and proportional to the number of model parameters included in the layer.

Figure 3-8 shows this linear relationship. Note that the loading time of the *Logits* layer, the one immediately following the *Transformer* block, is approximately 6× lower than transmitting the activation output of the transformer block using a 10 Gbps network. The gap will be larger when the training occurs in a decentralized environment with wide-area networks. With the forthcoming PCI-e 5.0 protocol and an expected higher PCI-e bandwidth, the gap between sub-model loading and activation transmission will likely widen.

3.3.3 Streamlining Sub-model Swapping

Although our profiling results clearly show advantages of memory swapping over network transmission, building a model swapping schedule to seamlessly overlap model execution and loading is challenging. Figure 3-9 and Figure 3-10 show the layer-by-layer breakdown of forward and backward propagation time, respectively. The figures suggest that backward propagation time is approximately 3× of the corresponding forward propagation time for each layer. Furthermore, the backward propagation times are comparable to the loading times in Figure 3-7, opening up opportunities for overlapping sub-model execution and loading during the backward pass. The first challenge is that the forward propagation times are consistently shorter than the corresponding layer loading time, which stalls model execution waiting for the next sub-model to be loaded. To address this issue, Atom prolongs the execution times of forward propagation by feeding multiple mini-batches to the forward pass before the backward pass starts. Such a gradient accumulation technique is commonly used in asynchronous training and helps maintain a high computation-to-swapping ratio in the forward pass.

A more difficult challenge is to deal with layers that have significant loading time yet trivial computation time. Layer *embedding* is one such example. It incurs nontrivial loading time due to its size (as shown in Figure 3-7) but negligible execution time (not shown in Figure 3-9 and Figure 3-10). Figure 3-11 shows the execution time of layer embedding relative to those of other layers in a transformer block. As shown in the figure, layer embedding's contribution to transformer' execution time is negligible in both forward and backward propagation and its percentage decreases as model size increases. The embedding layer encodes an input word from a sentence into a numerical vector featuring the word in the context. It is always placed in front of a transformer block right after a training step starts. The problem with caching the embedding layer on a GPU is that it increases memory usage, while using the CPU to run the embedding (∼2.4 GB, 50K vocal, d_model=12288, FP32) on

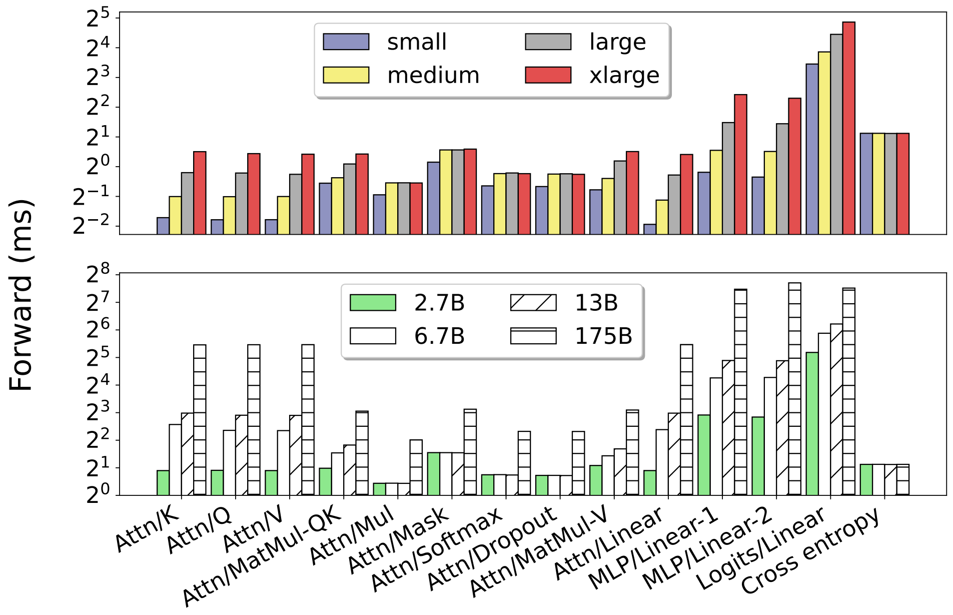


Figure 3-9: Breakdown of forward propagation time in GPT-3.

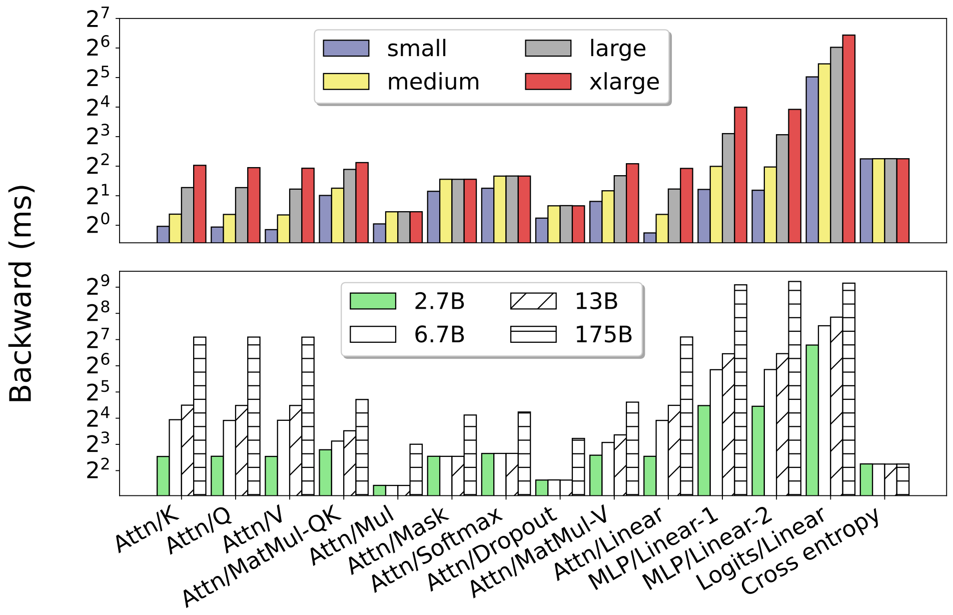


Figure 3-10: Breakdown of backward propagation time in GPT-3.

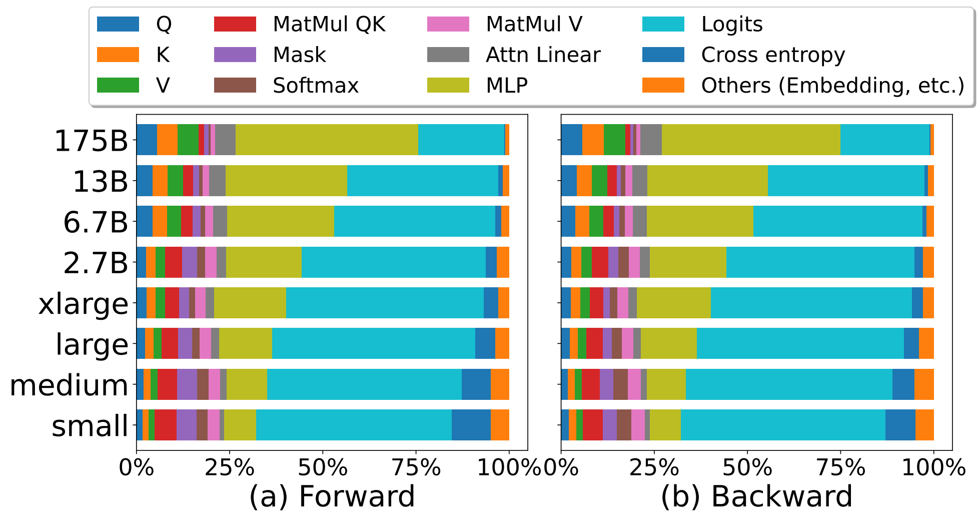


Figure 3-11: Time breakdown of operators in a transformer block, including the execution time on the *Embedding* layer. GPT3-175B a reduced size of 2-layer subset.

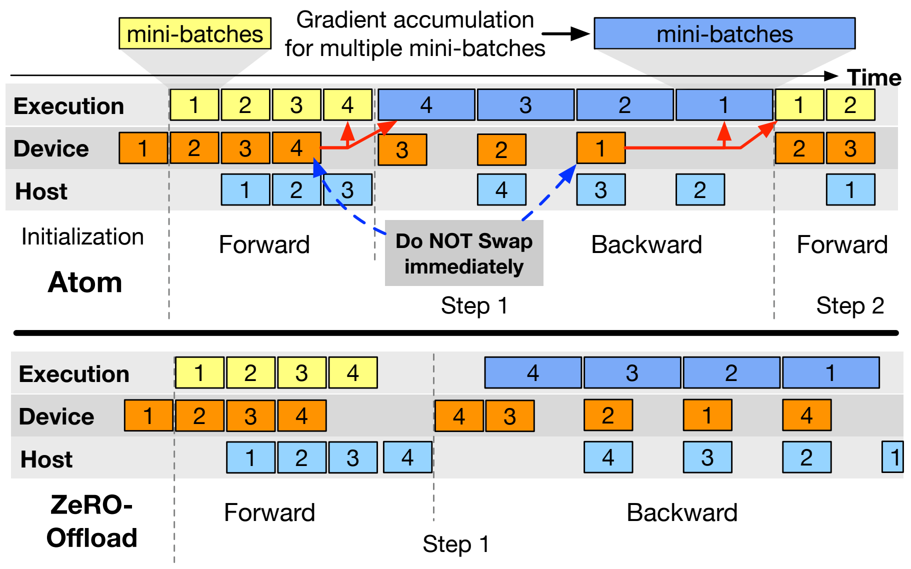


Figure 3-12: Schedule of sub-model swapping. Sub-models are denoted by numbered blocks. The *embedding* layer is included in sub-model 1. ATOM demonstrates improved efficiency compared to the ZeRO-Offload schedule.

resource-constrained GPUs can lead to slow performance.

Figure 3-12 shows a schedule of sub-model swapping in ATOM . The dotted vertical lines separate training steps, each containing a forward (yellow) and backward (blue) pass. The *execution* line shows the timeline of sub-model execution and the *device* and *host* lines show when the sub-models are placed in device memory and host memory, respectively. As shown in Figure 3-12, an advantage of model swapping

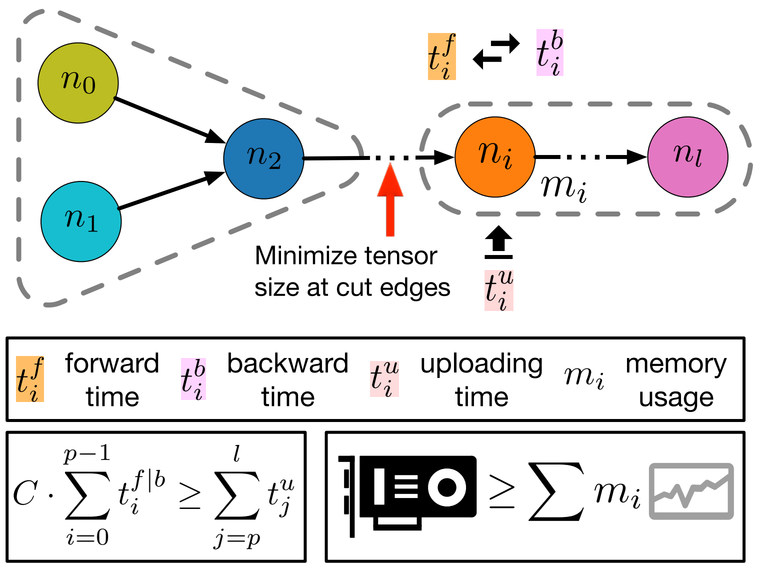


Figure 3-13: The formulation of model partitioning.

in LLM training is that there exists important locality: the last sub-model in the forward pass is immediately executed in the backward pass; the last sub-model in the backward pass is the first needed in the forward pass of the next training step. Since the embedding layer belongs to sub-model 1 and it does not need to swapped out at the end of the backward pass, it will be readily available when the next forward pass begins. This eliminates the potential GPU idleness due to the mismatch between the execution time and loading time in the embedding layer. Figure 3-12 also shows gradient accumulation from multiple mini-batches ensures that the execution time of any layers in the forward and backward pass exceeds their corresponding loading time to avoid GPU idleness.

3.3.4 Model Partitioning and Code Generation

The key to build a model swapping schedule that seamless overlaps model execution and loading is to determine a model partitioning scheme that satisfies that 1) any sub-model fits in the GPU device memory; 2) adjacent sub-models in the topological order in the model's computation graph have comparable execution and loading time. To guide model partitioning, we augment the computation graph with operator/layer's forward/backward execution time, loading time, and memory usage. To collect each operator's information, the model is profiled offline, in which the computation graph is executed operator (layer) by operator (layer). Profiled operators are swapped out to host memory to ensure that even the largest GPT-3 175B model can be profiled on a single GPU. We devise an heuristic exhaustive search algorithm to find a valid partitioning that satisfies all constraints.

Problem definition. A neural network can be represented as a directed computation graph (DAG) $G(E, V)$, where each node $v \in V$ is a layer or an operator, and an edge $e \in E$ is an intermediate input or output tensor to the neighboring nodes. The graph partitioning problem can be expressed as an optimization problem subject to multiple constraints, as shown in Figure 3-13. We design an algorithm to automatically partition the model in Atom subject to constraints, including the computation-to-loading ratio between adjacent sub-models, GPU memory capacity, the minimization of tensor size at the cutting of edges. Node ID i is sorted in the topological order of the computation graph and indexed from 0 to l. Attribute information of each node includes the max working memory size m_i at runtime, the forward execution time t_i^f, the backward execution time t_i^b, and the loading time t_i^u from host memory to device memory.

The *objective function* is to minimize the computation time, i.e., the sum of all sub-model forward and backward propagation time, and to minimize the partitioning cost, i.e., the total size of intermediate tensors (the size of cutting edge) across different partitions. One *constraint* is GPU memory capacity. The mainstream commodity GPU memory size is 11 GiB, 16 GiB, or 32 GiB. Another constraint the computation/prelading ratio where the partitioning result should ensure that the current sub-model computation time can overlap the next sub-model layer preloading time. Given a bipartite model partitioning $\{n_0, \ldots, n_i\}$ and $\{n_{i+1}, \ldots, n_l\}$, it should satisfy the condition that both $\sum_{i=0}^{p} m_i$ and $\sum_{k=i+1}^{l} m_k$ do not exceed the GPU memory capacity. Besides, computation can overlap with next sub-model loading $C \cdot \sum_{k=0}^{i-1} t_k^{f|b} \geq \sum_{k=i}^{l} t_k^o$, where C is a constant to specify the degree of gradient accumulation in order to match the execution time of forward propagations with their sub-model loading time. We empirically determine C offline via profiling for a

Algorithm 1: Partitioning a GPT-3 model.

```
1: function ValidConstraints($G, c_s, c_e, l_s, l_e$)
2:     if $G.mem(c_s, c_e) \leq GPU\_capacity$ // pruning
          $G.mem(l_s, l_e) \leq GPU\_capacity$
          $G.comp\_t(c_s, c_e) \geq G.load\_t(l_s, l_e)$ then
3:         return True
4:     else
5:         return False
6: end if
7: end function
8: // $c_s$ and $c_e$ are the begin and end operator indexes considered for sub-model
   execution; $l_s$ and $l_e$ are operator indexes for sub-model preloading; the objective
   is to match the computation time between $c_s$ and $c_e$ with the loading time between
   $l_s$ and $l_e$.
9: function PartitionModel($G, c_s, c_e, l_s, l_e, t, partitions$)
10:     if $!ValidConstraints(G, c_s, c_e, l_s, l_e)$ then
11:         return
12:     end if
13:     if $l_e = G.num\_nodes - 1$ len($t$) $\geq 0$ then
14:         $partitions$.insert($t$)
15:         return
16:     end if
17:     // squeeze boundary to keep more nodes within a certain range
18:     for $\hat{l_e} \leftarrow (G.num\_nodes - 1$ to $l_e$, $step\_size)$ do
19:         $t$.insert($(l_s, l_e), (l_e + 1, \hat{l_e})$)
20:         // recursively search for a valid partition
21:         $PartitionModel(G, l_s, l_e, l_e + 1, \hat{l_e}, t, partitions)$
22:         $t$.pop($(l_s, l_e), (l_e + 1, \hat{l_e})$) // backtracking
23:     end for
24: end function
25: function Main($G$) // graph G is topologically sorted
26:     $c_s, t, partitions \leftarrow 0, [], []$ // partial and final results
27:     // search valid partitions of the computation graph
28:     for $c_e \leftarrow G.num\_nodes - 2$ to $c_s$; $l_s \leftarrow c_e + 1$ do
29:         for $l_e \leftarrow (G.num\_nodes$ to $l_s$, $step\_size)$ do
30:             $PartitionModel(G, c_s, c_e, l_s, l_e, t, partitions)$
31:         end for
32:     end for
33: end function
```

particular GPU.

Model partitioning algorithm. Based on the augmented computation graph, we design an exhaustive layer-wise search algorithm to find a feasible partitioning that satisfies all constraints. Algorithm 1 shows the pseudo code for a feasible partitioning. The input to the algorithm is the computation graph G of a GPT-3 model that is sorted according to the nodes' execution order on GPU. The output is a candidate partitioning stored in *partitions*. The algorithm recursively evaluates the last edge of a computation graph (line 9) to see if a partitioning that generates the largest sub-model given the computation graph satisfies the constraints. The sub-model evaluated for computation is indexed by c_s and c_e while the sub-model tested for its loading time is indexed by l_s and l_e. The search is performed with a *step_size* of 1. Graph partitioning is an NP-hard problem and an exhaustive search could be prohibitively expensive when the graph contains hundreds of thousands of nodes. Knowledge on the structure of GPT-3 helps accelerate the search and can be input to the algorithm as an additional constraint. GPT-3 contains identical transformer blocks and this domain knowledge greatly reduces the search space. Among the multiple feasible partitions returned by the algorithm, Atom selects the one that minimizes the total tensor sizes between sub-models.

The time complexity of the given algorithm is exponential in the number of nodes in the input graph G, and also depends on step size of the nested for-loops in the main function. The time complexity is determined by the number of recursive calls made in the *PartitionModel* function, which is a combination of the number of nodes in G and the step size. The complexity of the *ValidConstraints* function is constant as it is a simple comparison of memory and computation time between two sub-models. The complexity of the Main function is determined by the number of recursive calls made by the *PartitionModel* function, as well as the number of iterations in the for-loops, which depend on the number of nodes in G and step size.

Model compilation and code generation. Atom does not require any changes to user code and automatically generates Python code for the partitioned model. The compiler runtime takes as input the augmented computation graph, the partitions returned by the search algorithm, and the original model (`torch.nn.Module`). The output of the code generation engine is Python source code of the partitioned with each sub-model being a separate class. The same Python source code is distributed to all Atom worker nodes in a decentralized network. The process of code refactoring can be automated.

3.3.5 Model Distribution and Fault Tolerance

Node failures and transient network disruptions are common in a large-scale decentralized network. Unlike model and parallelism that suffer from single point of failure or do not support dynamic node join/leave, Atom allows volunteer nodes to independently train a copy of the complete model and relies on an periodic allreduce communication to synchronize copies of the model, in a way similar to data parallelism. Atom employs a distributed hash table (DHT) to monitor the status of volunteer nodes. Each node periodically publish its status to the DHT via a heartbeat message. Among the information included the heartbeat, nodes report the number of mini-batches they have processed so far. Atom summarizes the total number of mini-batches processed by all nodes and triggers an allreduce communication if a global batch size has been reached. In cases when node failure or departure occurs, training can still proceed as long as other nodes are still able to process mini-batches and compensate the loss due to the leaving node.

The shared GPU clusters contributed by volunteers can handle multiple training jobs simultaneously given enough host memory. Management of different training jobs submitted by volunteers require individual progress of their training jobs. In a shared environment with different training jobs, each job will be scheduled for a timeslot to execute. Participants publish the generated model code and store the model to the distributed hash table so the model can be retrieved without any central storage even volunteer hardware continuously join, leave, and fail. Also, the compiled model can be reused for the same hardware configuration. Therefore, Atom can coordinate multi-tasking globally by means of service orchestration which is a future direction.

3.4 Implementation

Atom is implemented based on the PyTorch framework with ∽4K lines of code in Python. It uses the DHT implementation in Hivemind [118] to coordinate volunteer nodes in a decentralized network and the code generation engine in FT-Pipe [38] to generate code for sub-models. The system consists of model tracing, profiling, model partitioning, and compilation. In profiling, a model is analyzed with a sample input and operators and layers are traced to construct a computation graph in the form of an internal representation. We measure the forward and backward step time, and loading time from CPU to GPU of each operator and layer by `torch.cuda.Event.elapsed_time`, and the timing runs are repeated multiple times. We measure the memory consumption of each layer by `torch.cuda.max_memory_allocated`.

In the partitioning phase, we apply Algorithm 1 to assign partition id for each node based on the profiling statistics in the computation graph. At code generation phase, the compiler populates the initial function to define operators and layers of a sub-model, and define the execution flow in the `torch.nn.Module.forward` function base on the computation graph. The input shape is derived from the cutting edge between sub-models. During model execution, Atom maintains two CUDA streams. One is responsible for executing the current sub-model and the other one asynchronously prefetches the next sub-model immediately after the current sub-models starts. The former stream also swaps out a completed sub-model to host memory. Note that Atom does not swap out the last sub-model at the end of the forward pass or the first sub-model at the end of the backward pass to exploit locality between the forward and backward propagation.

3.5 Evaluation

In this section, we seek to answer the following questions:

- How well do model swapping perform on a single GPU in training LLMs compared with baseline pipeline parallelism strategies across servers in a low-bandwidth network environment? (§ 3.5.2)

- How well does Atom scale with an increasing number of volunteer nodes? (§ 3.5.2)
- Does decentralized training in Atom affect the convergence of the trained model in the presence of node join and leave (fault tolerance) ? (§ 3.5.2)

3.5.1 Evaluation Methodology

Experimental setup. We use three types of GPU servers coded *high*, *medium*, and *low*. High is configured with four NVIDIA Tesla V100 GPUs, 72 cores (dual Intel Xeon Gold 6140 CPU@2.30GHz), and 385 GB host memory. Medium is configured with four GeForce GTX 1080 Ti GPUs, 72 cores (dual Intel Xeon CPU E5-2695 v4@2.10GHz), and 256 GB host memory. Low is configured with four GeForce GTX 1080 GPUs, 40 cores (dual Intel Xeon Silver 4114 CPU@2.20GHz), and 256 GB host memory. All servers have PCI-e 3.0x16. The CUDA Version is 11.3, and the PyTorch version is 1.11.0.

Workloads. We mainly focus on GPT-3 models with various configurations [21], including GPT-3 Small, Medium, Large, XL, 2.7B, 6.7B, 13B, 175B. Note that Atom requires that a model can fit in host memory. The full version GPT-3 175B model needs 2.8 TB memory and does not fit in our platforms. In general, GPT-3 175B can be trimmed down by removing a few identical transformer blocks to fit in our 385 GB host memory. For fair comparison with pipeline parallelism, in which the model size cannot exceed the aggregate GPU device memory, we evaluate a GPT-3 175B model with two transformer blocks and a model size of 68 GB.

Baseline. For each GPT3 model configuration, we compare Atom against Petals [18] based on BigScience [55] Project BLOOM [108] by using the schedule policy of GPipe [54] and PipeDream [78]. The baselines were also implemented in Hivemind [118] similar as Atom to account for the effect of software architecture on training performance. Other state-of-the-art approaches are *not* directly comparable since many assume the availability of high-speed interconnect. For example, ZeRO-

Offload [117] has comparable performance of a single GPU swapping scheduling but it distributes optimization states, gradients, and parameters across GPUs, which can be inefficient over low-bandwidth networks. This is a significant contrast to our approach, which utilizes distributed decentralized training to avoid single-point failure. FTPipe [38] achieves high GPU utilization by aggressively packing the pipeline with mini-batches and relies on the PCI-e bus for data transmission between sub-models. In contrast, ATOM focuses on LLM training over low-bandwidth networks. Similar approaches that exploit intra-layer parallelism, such as Megatron-LM [112], require even higher bandwidth.

Evaluation metrics. We use per result GPU time, defined as the time to process one mini-batch per GPU, to measure training performance. Practically, we measure the number of mini-batches processed per GPU per unit time and take the reciprocal. We are also interested in the total time required to finish one training step, including the forward and backward propagation time, allreduce communication time, and the optimization time. We use the allreduce time to study ATOM's scalability.

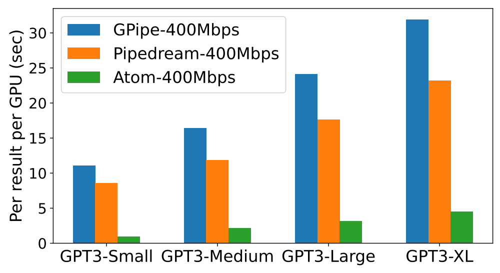
3.5.2 Experimental Results

Table 3.3: The configuration to compare training performance.

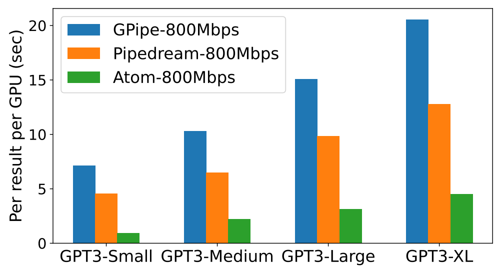
Setting	Configuration
Schedule policy	Petals (GPipe, PipeDream), ATOM
Bandwidth	400 Mbps, 800 Mbps, localhost
Model config	GPT-3-Small, Medium, Large, XL, 2.7B, 6.7B, 13B (18 layers), 175B (2 transformer blocks)
GPUs	4×1080 (8 GB), 4×1080Ti (11 GB), 4×V100 (32 GB)

Training performance

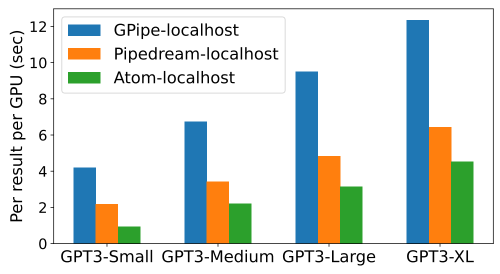
We first compare per result GPU time due to ATOM and the two baselines, GPipe and PipeDream, with three network bandwidth settings: 400 Mbps, 800 Mbps, or localhost, the hypothetical upper bound on bandwidth. We throttled the bandwidth



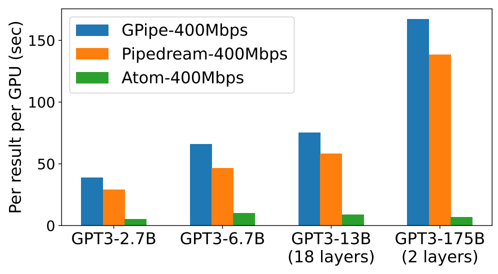
(a) 1080 Ti 400Mbps



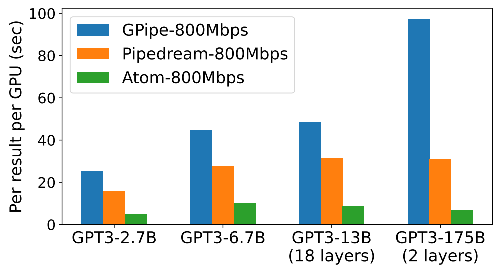
(b) 1080 Ti 800Mbps



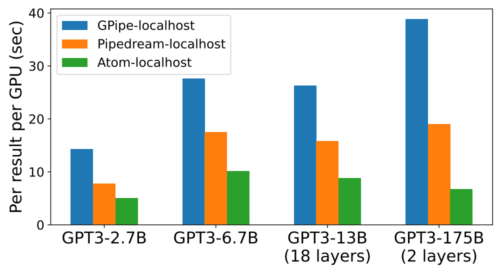
(c) 1080 Ti Localhost



(d) V100 400Mbps



(e) V100 800Mbps



(f) V100 Localhost

Figure 3-14: Comparison of per result (mini-batch) GPU time (w/o allreduce and optimizer step).

using Wonder Shaper [15]. The configurations are summarized in the Table 3.3. In this experiment, we disabled allreduce communication and the optimization step to focus only on the forward and backward passes. In all experiments, we used four GPUs on a single host without communications between hosts. This controlled environment helps compare the efficiency of the training pipeline in the baselines with that in ATOM. Models are optimally partitioned on four GPUs with minimal inter-sub-model tensor transmission for the baselines. ATOM uses the automatically partitioned models by its search algorithm. We set the degree of gradient accumulation to 4 mini-batches.

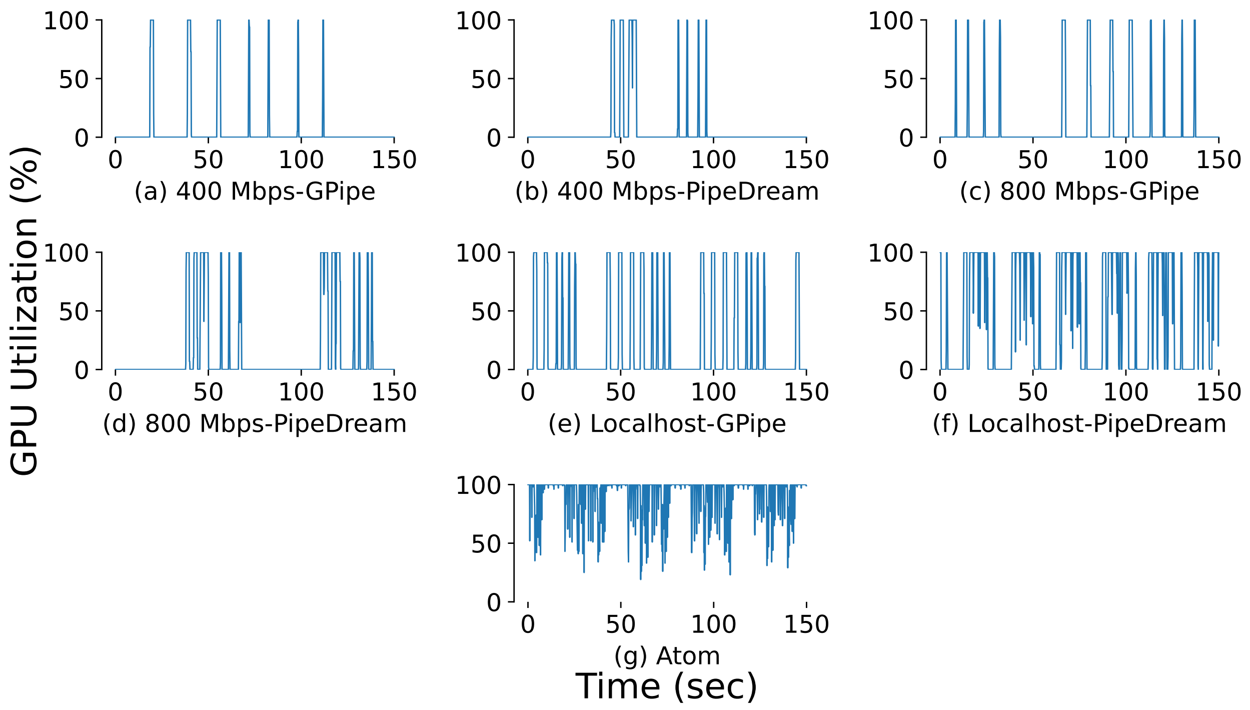


Figure 3-15: GPU utilization over different network bandwidth of GPT-3 175B (2 transformer blocks) by *nvidia-smi*.

Figure 3-14 shows the averaged step time (both forward and backward) to process a mini-batch per GPU. Among the three approaches, Atom significantly and consistently outperformed GPipe and PipeDream in all experiments. The gap between Atom and the baselines widens as model size increases or the bandwidth decreases, suggesting that Atom is most suitable for training large-scale models with low-bandwidth networks. PipeDream also consistently outperformed GPipe due to its asynchronous training pipeline that is able to overlap the processing of multiple mini-batches.

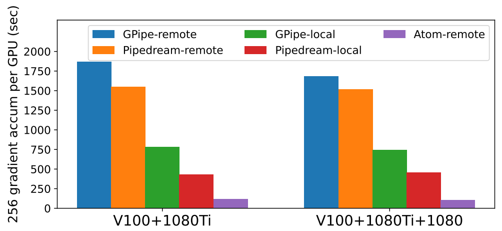
To further study the causes of differences in training performance, we plot GPU utilizations over the same period of time for Atom , GPipe, and PipeDream in Figure 3-15. Both GPipe and PipeDream suffered from substantial idleness in GPU across all experiments. The idleness increases as network bandwidth drops. It suggests that even with multiple mini-batches being fed to the pipelines, computation cannot overlap with and hide inter-model communications, causing bubbles in the pipeline. In comparison, Atom is not affected by network speed and the automatically generated model partitioning helps streamline model execution and loading, leading to high GPU utilization. Both GPipe and PipeDream achieved the highest GPU utilization of 18.3% and 46.3%, respectively. with the localhost network. On contrary, Atom achieved a GPU utilization of 91.9%, almost twice as much as PipeDream with an asynchronous pipeline.

Scalability

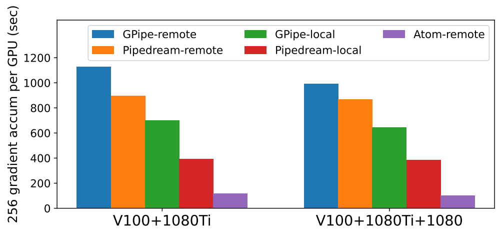
The scalability in distributed training is usually determined by the allreduce communication phase for model synchronization. For pipeline parallelism, there is a limit on how much a model can be partitioned to maintain a high computation-to-communication ratio. To scale to a large number of GPUs, a common practice is to have independent pipelines, each spanning multiple GPUs and servers. Similar to data parallelism, the pipelines are periodically synchronized via allreduce communication. To study Atom's scalability, we focus on how is Atom's performance compared with the baselines when using the same number of GPUs and the changes in its allreduce time as the number of GPU increases.

Due to no access to a large GPU cluster, which motivated this work, we placed the three GPU servers, i.e., *high*, *medium*, and *low*, into three subnets connected by campus Internet to emulate a decentralized network. We configured the baselines with two settings and two scales. Suffix *local* indicates GPUs on the same host can communicate with unlimited bandwidth while *remote* indicates the bandwidth between any GPUs are throttled to emulate a wide-area network. We evaluated a 2-pipeline and 3-pipeline setting for the baselines. For example, the 2-pipeline experiment ran one pipeline on the V100 and 1080Ti server, respectively. For all experiment, Atom was configured to use the same type and number of GPUs as the baselines. All approaches were configured to perform an allreduce communication once a global batch size of 256 is reached.

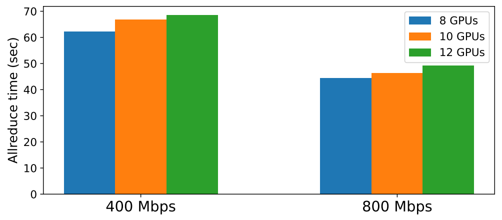
Figure 3-16a and Figure 3-16b show the comparison of the average training time that is needed to finish one global batch, i.e., 256 mini-batches. Note that in this experiment allreduce communication and optimizer updates were enabled. Again,



(a) 400 Mbps.



(b) 800 Mbps.



(c) Change of allreduce time in Atom .

Figure 3-16: Comparison of total training time among GPipe, PipeDream, and Atom under different bandwidth.

Atom clearly outperformed the other two approaches by a large margin. It indicates that Atom's decentralized architecture built on GPUs training on independent models is more scalable than the pipeline architecture that requires frequent inter-machine communication. Figure 3-16c shows the change in Atom's allreduce time as the number of GPU increases. The results suggest that there is no dramatic change in the allreduce time as the system scaled. Because Atom consists of largely independent GPUs, its scalability is determined by the selection of an allreduce communication algorithm, which is orthogonal to Atom's design.

Effectiveness of training and fault tolerance

Model convergence. To improve training efficiency, Atom differs from the two baselines in training method. GPipe uses a synchronous pipeline that suffers from low GPU utilization but offers fast model convergence. PipeDream uses an asynchronous pipeline to overlap the training of multiple mini-batches. It improves GPU utilization but allows for updates on stale model parameters that results in slower convergence. In Atom , gradient accumulation is necessary for prolonging the forward pass to

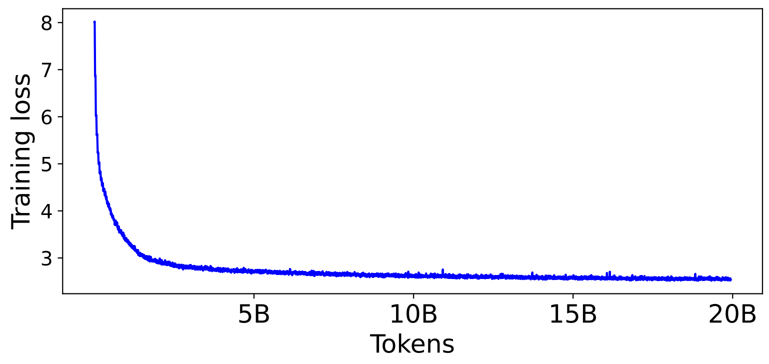


Figure 3-17: Convergence of a GPT-3 Small model in ATOM.

match its model loading time. However, gradient accumulation can negatively affect convergence. To verify ATOM's convergence, we trained a GPT-3 small model from scratch using the Wikipedia dataset [126] until convergence and compare that with the baselines. The training used a learning rate 1×10^{-4}, 300K total training steps with a linear warmup of 3K steps. We used the CPU AdamW [71] optimizer ($\beta1 = 0.9$, $\beta2 = 0.999$). We set the target global gradient accumulation size to 512 samples. The target group size to do the model averaging (ring-allreduce step) and optimization is 12. The total number of GPUs is 12 from 3 servers: 4×V100, 4×1080Ti, and 4×1080. Figure 3-17 shows the training loss over 20 billion tokens. We observe that the training loss steadily decreased and eventually converged to an accuracy comparable to the published result. We conclude that ATOM's architecture design for training efficiency does not undermine training effectiveness.

Fault tolerance. During the experiment, we also deliberately killed two to four GPUs to emulate dynamically node joining/leaving. The training did not experience a disruption and was able to complete, though training performance dropped.

3.6 Conclusion

We present ATOM , a system for asynchronous training of massive models in a decentralized environment. The key insight that motivated ATOM design is that huge LLMs can be executed on a single GPU layer by layer via memory swapping. ATOM addresses the overhead of memory swapping by deriving an optimal schedule of model swapping through detailed profiling of individual layers of an LLM. The overarching objective that guides Atom design is to avoid GPU idleness as much as possible. Through comprehensive experiments, we demonstrate that the loosely-coupled distributed training architecture is advantageous than the tightly-coupled pipeline parallelism. We acknowledge that individual servers still needs to equip sufficient host memory to accommodate LLMs. We envision that with the development of the compute express link (CXL) interconnect, host memory will be more accessible than accelerators and high-speed networks.

Chapter 4

Accelerating Data Preprocessing in Deep Learning: An Empirical Study on the PerFect Data Reuse Method

As deep learning (DL) models grow in complexity and scale, data preprocessing has emerged as a critical stage in the pipeline. However, the CPU's processing speed can often become a bottleneck for downstream training processes, despite the increasing power of GPUs. To mitigate this challenge, researchers have proposed various solutions, including data echoing with or without augmentation, specialized data loading libraries such as NVIDIA DALI, and additional hardware devices. However, these approaches have limitations, such as the need for additional GPU memory with DALI and difficulty achieving the desired level of accuracy with data echoing. In this chapter, we present a novel method, named *PerFect*, to improve the throughput performance of the data preprocessing stage and achieve the desired accuracy when reusing cached data. Our proposed approach involves pre-training the model using cached data for most epochs, followed by fine-tuning to achieve the desired accuracy. This method avoids the limitations of data echoing approaches without the need for additional hardware or third-party libraries. We provide theoretical insights to analyze the problem and evaluate our approach on both CIFAR10 and ImageNet datasets using various models. Comprehensive experiments demonstrate the efficacy of *PerFect* in achieving the target accuracy while reducing training time. As such, our proposed method provides a promising avenue for accelerating the data preprocessing stage in deep learning pipelines.

4.1 Introduction

The exponential growth of Big Data has empowered the success of Deep Learning (DL) applications, which have become a prevalent approach for a wide range of tasks [65]. For instance, DL has shown outstanding performance in image classification [50], natural language understanding [21], and drug discovery [62]. However, training DL models requires significant computing resources, and one of the critical stages in this process is the data preprocessing stage. The data preprocessing stage is responsible for loading and parsing input data and preparing it for a DL model by applying data augmentations [113].

Despite its importance, the data preprocessing stage can become a bottleneck in the overall DL workflow [48, 76]. This is mainly due to the fact that data preprocessing is primarily performed on CPUs, whereas GPUs are becoming more and more powerful due to Moore's law. When dealing with massive datasets, the problem becomes even more pronounced, as the data preprocessing stage can require extensive computational resources.

To address this issue, several approaches have been proposed. While these approaches have demonstrated success in enhancing DL system performance, they also have limitations. One approach is to use specialized hardware accelerators like GPUs or FPGAs to offload data preprocessing tasks. The NVIDIA Data Loading Library (DALI) is an example of such an approach . DALI [39] allows users to transfer all or part of the data preprocessing workload from the CPU to the GPU, thereby reducing the total training time. However, DALI requires additional GPU memory which may be limited compared to CPU memory. DLBooster [26], for instance, relies on the availability of extra hardware like FPGAs which may not always be feasible. The second approach is to utilize algorithms that make use of idle accelerator cycles to

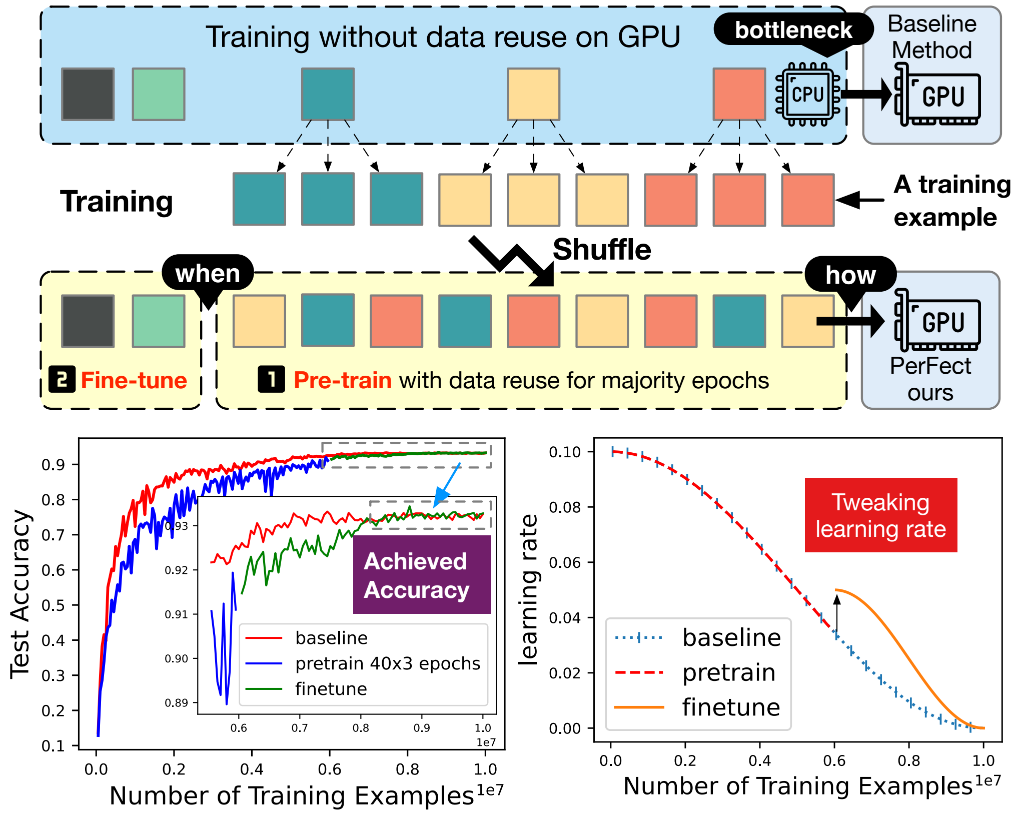


Figure 4-1: The effect on model performance with *PerFect* data reuse.

boost the efficiency of data preprocessing. One such algorithm is data echoing [29], which stores a copy of the data on the accelerator and reuses it during idle periods, reducing data transfer between the CPU and accelerator and improving DL system performance. However, data echoing and partial data reuse methods cannot meet desired accuracy requirements even with partial data reuse [66]. A recent study proposed a third approach MinIO cache that involves caching sharded datasets and fetching them through a high-speed network to enhance DL system performance [76]. Network latency in MinIO cache can slow down the overall system performance and increase response times for users. Moreover, caching may not scale well as the dataset size grows, leading to prohibitive memory requirements and increased costs for maintaining a high-speed network as the number of nodes increases. Overall, while these solutions have shown promise in enhancing data preprocessing performance, they also have limitations that must be considered when selecting the appropriate solution for a particular use case.

Table 4.1: Summary of key takeaways, findings and implications of pre-training reusing input data and fine-tuning.

Considerations	Finding & Implications
Determining the optimal timing for fine-tuning	Initiate fine-tuning after observing a plateau in the model's performance during validation.
Impact of shuffle buffer size	Larger shuffle buffer sizes make it easier to reach the target accuracy due to randomness.
Before and after data augmentation	Data augmentation can speed up convergence and assist with fine-tuning.
Impact of repetitive times	Fewer repetitive times make the fine-tuning process easier.
Learning rate scheduler	Higher learning rates can compensate for shifted data distribution.
Hyperparameter tuning	The choice of hyperparameters significantly impacts a model's performance.

In this chapter, we introduce *PerFect*, a novel method to improve the accuracy performance of the data preprocessing stage in DL systems while retaining the original accuracy targets, without the need for additional accelerators. Our approach consists of a two-phase training process: pre-training and fine-tuning. During the pre-training phase, we train the model for a majority of the epochs using data reuse, enabling the model to learn general features. In the fine-tuning phase, we refine the model without data reuse, allowing the model to learn task-specific features through training on data that is more relevant to original data distribution, thereby achieving the desired accuracy performance. As depicted in Figure 4-1, we first pre-trained the model for 40 epochs, repeating the process three times, for a total of 120 pre-training epochs. We then fine-tuned the model for an additional 80 epochs, bringing the total number of training epochs to 200. The training was conducted using a batch size of 2048 and a shuffle size of 2048 on 4 GPUs. The results demonstrate that applying data reuse three times increased the testing accuracy to 93.25%, matching the baseline, while decreasing the decoding time by 40% in pre-training. This emphasizes the efficacy of data reuse in enhancing model performance and optimizing training efficiency.

Although the concept of fine-tuning after pre-training may seem straightforward, it presents significant challenges in determining the optimal approach. In addition, we provide other examples and case studies, which we will consider in Table 4.1. This work aims to address several key questions related to fine-tuning, including determining the appropriate time to initiate pre-training, evaluating the benefits and drawbacks of initiating pre-training and fine-tuning earlier or later, identifying the most effective method for pre-training the model either through immediate data reuse without augmentation or through data augmentation for reused samples, analyzing the influence of buffer size and data shuffling on fine-tuning, and assessing the impact of repeating data samples on final accuracy. Our ultimate goal is to provide an empirical study and theoretical insights to better understand this problem and guide future research in this area.

In this study, we present a novel method called PerFect (Pre-train-Fine-tune) to enhance the data preprocessing stage in deep learning systems through a two-stage training approach. We evaluate the effectiveness of our approach on the CIFAR10 and ImageNet datasets, using various deep learning models, and demonstrate that it achieves the target accuracy while decreasing training time. Our approach outperforms the existing data echoing method [29] in improving the overall performance of deep learning systems while achieving the target accuracy. The contributions of our study can be summarized as follows:

- We propose a novel method, PerFect, that enhances the data preprocessing stage in deep learning systems through a two-stage training approach. This approach provides theoretical insights to analyze the overfitting problem and tweaks learning rate and hyperparameters to overcome the issue.
- We discuss the metrics used to determine the transition to the fine-tuning stage when the model reaches the target accuracy. We address the challenges of determining the appropriate learning rate scheduler and hyperparameters for training the model and propose a method to resolve these issues.
- We compare our approach to the existing data echoing method and demonstrate its superiority in improving the overall performance of deep learning systems while achieving the target accuracy.
- We provide a rigorous evaluation of our method on both the CIFAR10 and

ImageNet datasets, using representative deep learning models, and demonstrate its effectiveness in achieving the target accuracy.

In conclusion, our study presents a novel method for enhancing the data preprocessing stage in deep learning systems, which improves their overall performance while achieving the target accuracy. Our approach provides theoretical insights and practical guidance for overcoming the overfitting problem and determining appropriate hyperparameters and learning rate schedulers.

4.2 Background

4.2.1 The Impact of Data Preprocessing Overhead on Deep Learning Systems

The data preprocessing stage is a crucial component of the deep learning workflow, as it prepares the data for input to the model. This stage involves ETL (extract-transform-load) operations such as loading and parsing (decoding) input data, applying data augmentation, and converting the data into a form suitable for input to the model. Data augmentation is used to increase the amount of training data by applying transformations such as rotation, scaling, and cropping to the input data. These transformations help to prevent overfitting, which occurs when the model is too closely fit to the training data and is unable to generalize to new data.

However, the data preprocessing stage can be a major bottleneck in the deep learning workflow, especially when working with large datasets or lightweight deep learning models. To investigate the impact of data preprocessing on deep learning performance, we conducted an experiment using one to twelve data loader workers to preprocess the ImageNet dataset for three deep learning models: ResNet18, ResNet50, and ResNet152. We used a testbed server equipped with four NVIDIA Tesla V100 GPUs, each with 32 GB of memory, and a total of 72 vCPUs powered by an Intel(R) Xeon(R) Gold 6140 CPU running at 2.30GHz.

The results of our experiment, as depicted in Figure 4-2, indicate that the data

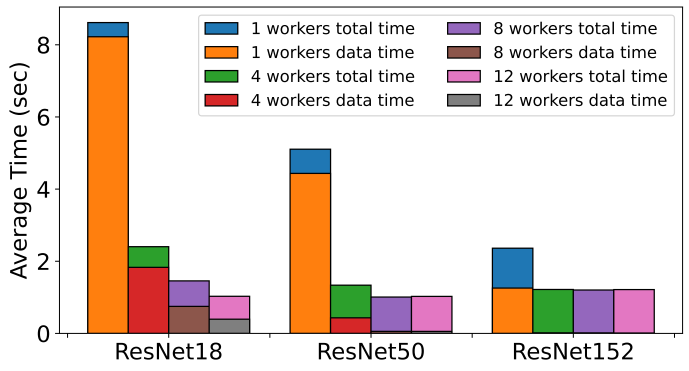


Figure 4-2: With decreased data workers causing increased loading time and lighter workloads accentuating the bottleneck in data processing.

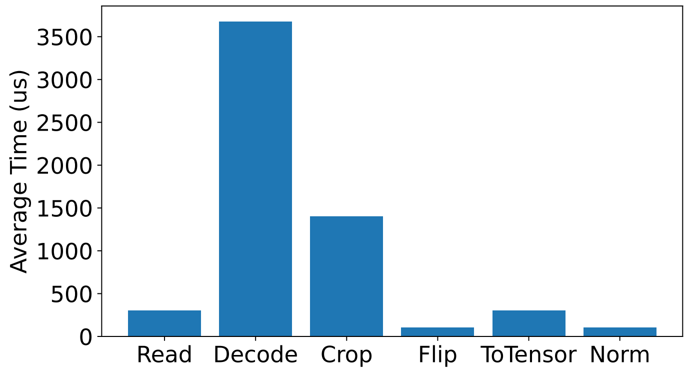


Figure 4-3: Preprocessing time per image for 1 worker using various transform operations, with decoding being the main contributor and data read from memory.

loading stage became increasingly pronounced as the size of the DL model decreased. The time taken for each image processing operation is shown in Figure 4-3. To maximize GPU memory usage, the batch size was set to 512 for ResNet18, 256 for ResNet50, and 128 for ResNet152. The images were processed using a varying number of workers from 1 to 12, with the `PyTorch` library and the `PIL` library utilized to load and decode the raw JPEG images into RGB format. The decoding step is a crucial part of the preprocessing process, but it can be reduced or eliminated by reusing data in the data buffer so we do not need to decode images every time.

The NVIDIA DALI library is designed to improve the efficiency of data preprocessing pipelines by offloading partial or all transformations to the GPU. As shown in Figure 4-4, the use of DALI requires additional GPU memory to process images for ResNet18. Specifically, the extra memory required for processing 512 images is 2916

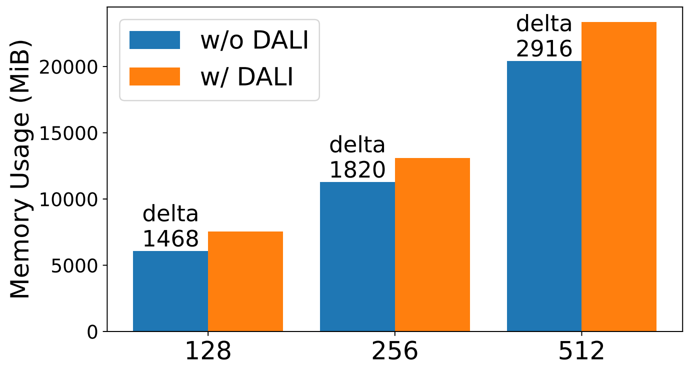


Figure 4-4: Memory usage comparison of NVIDIA DALI library vs PyTorch while processing various batch sizes for a ResNet18 model during training.

MiB, which is 14.28% more than the baseline. The memory required for processing 128 and 256 images is 1468 MiB (24.20%) and 1820 MiB (16.15%), respectively.

4.2.2 Deep Learning Training

Problem Definition. Let $\mathbf{x_i}$ be an input sample and y_i be the corresponding label in the training set $\{\mathbf{x_i}, y_i\}$, where N is the number of samples in the training set. The loss function, $\ell(p, y)$, measures the discrepancy between the model's prediction p for a given input $\mathbf{x_i}$ and the ground truth label y_i. The goal is to find the set of model parameters that minimize the average loss over the training set, as formulated in Eq.(4.1):

$$\mathbb{E}_{(\mathbf{x},y)\sim D}\left[\ell\left(f_\theta(\mathbf{x}), y\right)\right] := \min_\theta \frac{1}{N}\sum_{i=1}^{N}\ell(p(\mathbf{x_i};\theta), y_i) \tag{4.1}$$

where θ represents the model parameters, and D denotes the training dataset distribution.

To solve this optimization problem, we employ optimization algorithms such as stochastic gradient descent (SGD) or its variants. These algorithms iteratively update the parameters in the direction of the negative gradient of the expected loss with respect to the parameters. The iteration continues until a stopping criterion is met, such as convergence to a local minimum or reaching a maximum number of iterations.

$$\theta_t = \theta_{t-1} - \eta \nabla_\theta \ell(\theta_{t-1}) \tag{4.2}$$

where θ_t is the updated model parameters at iteration t and $\nabla_\theta \ell(\theta_{t-1})$ is the gradient of the loss function with respect to the model parameters θ at iteration $t-1$.

Weight decay is a regularization technique that helps to prevent overfitting by adding a term to the objective function that penalizes large parameter values. The weight decay hyperparameter λ determines the strength of this penalty. The objective function with weight decay can be expressed as:

$$\min_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(p(\mathbf{x_i}; \theta), y_i) + \frac{\lambda}{2} \|\theta\|_2^2 \tag{4.3}$$

where $\|\theta\|_2^2$ is the squared L_2 norm of the model parameters.

The update rule becomes:

$$\theta_t = \theta_{t-1} - \eta(\nabla_\theta \ell(\theta_{t-1}) + \lambda\theta_{t-1}) \tag{4.4}$$

Momentum is a technique that helps the optimization algorithm to escape from local minima and to converge faster. It does this by adding a fraction of the previous weight update to the current weight update. The momentum hyper-parameter μ determines the strength of this fraction.

To perform SGD with momentum, we initialize an additional momentum term $g_0 = \nabla_\theta \ell(\theta_0)$ and update it at each iteration using the following rule:

$$\begin{aligned} v_t &= \mu v_{t-1} + \nabla_\theta \ell(\theta_{t-1}) \\ \theta_t &= \theta_{t-1} - \eta v_t \end{aligned} \tag{4.5}$$

where θ, g, v and μ are the parameters, gradient, velocity, and momentum respectively.

To perform SGD with weight decay and momentum and update the parameters using:

$$\begin{aligned} v_t &= \mu v_{t-1} + (\nabla_\theta \ell(\theta_{t-1}) + \lambda\theta_{t-1}) \\ \theta_t &= \theta_{t-1} - \eta v_t \end{aligned} \tag{4.6}$$

4.2.3 Parameter Updating using Data Reuse

If we reuse data for τ times, the model parameters are updated:

$$\begin{aligned} v_{t,i} &= \mu v_{t,i-1} + (\nabla_\theta \ell(\theta_{t,i-1}) + \lambda\theta_{t,i-1}), \quad & i = 1, 2, \ldots, \tau \\ \theta_{t,i} &= \theta_{t,i-1} - \eta v_{t,i}, & i = 1, 2, \ldots, \tau \end{aligned} \tag{4.7}$$

where τ is the times of reusing data from shuffle buffer size.

4.2.4 Pre-training and Fine-tuning

Pretraining and fine-tuning are essential techniques in the field of deep learning that improve the accuracy and efficiency of machine learning models, especially when dealing with large datasets or transfer learning scenarios.

Pre-training involves training a deep learning model on a large and representative dataset, allowing the model to learn general features that can be applied to a wide range of tasks. This process makes the model more versatile and robust, which is crucial for real-world applications. Data echoing is a method that can be used during pre-training to speed up the process by duplicating data instances. Moreover, a pre-trained model can be saved and reused for future tasks, reducing the need for retraining from scratch and saving time and resources.

Fine-tuning is a technique that involves taking a pre-trained model and adjusting its parameters on a smaller, more specific dataset. This process allows the model to learn task-specific features that are not present in the original dataset used for pre-training. As a result, fine-tuning can significantly improve the performance of the model on the specific task at hand. Additionally, fine-tuning can speed up the training process since the model starts with a good set of initial weights that have already been learned during pre-training.

Overall, pre-training and fine-tuning enable the development of highly accurate and efficient models. Pre-training allows the model to learn general features that can be applied to a wide range of tasks, while fine-tuning helps the model to adapt to more specific tasks, improving its performance.

4.3 The Risk Reusing Data: Overfitting

Overfitting is a common challenge in machine learning, which arises when a model becomes too complex and fits the training data too closely, leading to poor generalization performance. In this context, we present a case study to illustrate how overfitting can occur for a simple model when reusing data and highlight the importance of fine-tuning in the final stage of training to improve the model's generalization ability.

Furthermore, we suggest several considerations that can help alleviate the effects of overfitting, which are outlined in Table 4.1 and are based on theoretical analysis. These considerations include carefully selecting an appropriate shuffle buffer size, applying data augmentation, adjusting the number of repetitions, learning rate, and hyperparameters. By following these guidelines, we can enhance the generalization performance of machine learning models and reduce the risk of overfitting.

Overfitting case study. The linear regression model under consideration is represented as:

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{4.8}$$

where x is the input data, and θ_0 and θ_1 are model parameters. To evaluate the model's performance, we conduct an experiment where we generate a set of 50,000 data points around the line $y = 7 + 3x + \epsilon$, where ϵ is a random noise term. Figure 4-5 shows the training data points and the linear regression model used for fitting. The cost function for this model is defined as the mean square error (MSE) between the predicted value and the actual value:

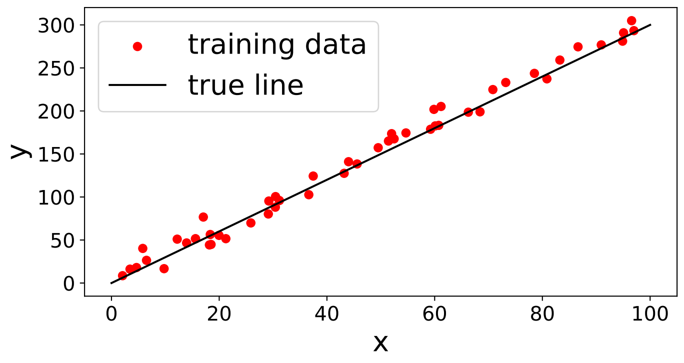


Figure 4-5: Linear regression example model used for fitting, along with the training data.

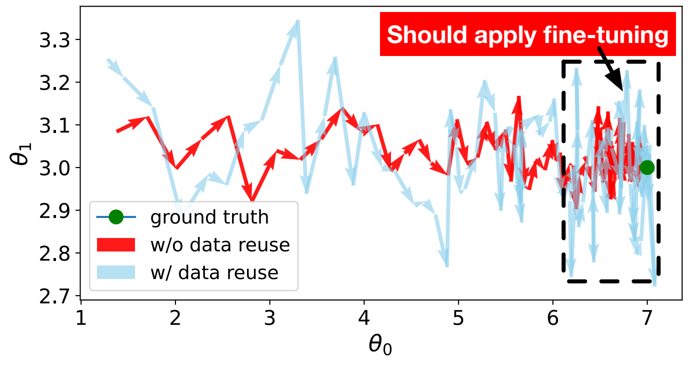


Figure 4-6: A comparison of the parameter updates of the model with θ_0 and θ_1, with and without data reuse.

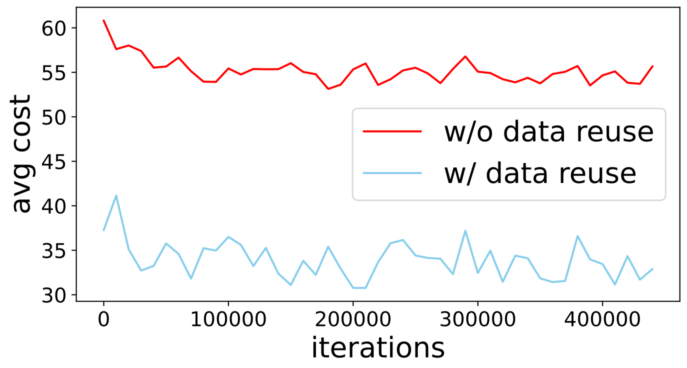


Figure 4-7: The comparison of the training loss between with data reuse and one without.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2 \tag{4.9}$$

We use stochastic gradient descent (SGD) with a learning rate of 5e−5 for optimization. Figure 4-6 displays the updates of model parameters θ_0 and θ_1 with and without data reuse to the true values $\theta_0 = 7$ and $\theta_1 = 3$. We observe that the variance of the parameters increases when data reuse is employed. The sum of all variable variances was 2.17 with data reuse, which is higher than the value of 1.85 without data reuse. Additionally, Figure 4-7 illustrates the training loss, showing that the training loss with data reuse is lower than that without due to overfitting to the training data.

Bias-variance tradeoff. To simplify the discussion, we consider the mean squared error (MSE) of the estimate $\widehat{X}$ of the true value X of a model as follows:

$$\mathrm{MSE}(\widehat{X}) = \mathbb{E}\left((\widehat{X} - X)^2\right) = \mathrm{Var}(\widehat{X}) + \left(\mathrm{Bias}(\widehat{X})\right)^2 \tag{4.10}$$

If we repeat the data reuse n times, the expected mean squared error at a single training example $\widehat{X}$ will increase n^2 times:

$$\mathrm{MSE}(n\widehat{X}) = n^2 \left(\mathrm{Var}(\widehat{X}) + \left(\mathrm{Bias}(\widehat{X})\right)^2\right) \tag{4.11}$$

Thus, reusing the same training data for each iteration can result in accumulating mean squared error for the entire dataset.

L^2 **Regularization and Repetitive Data**. One way to prevent overfitting is by applying L^2 regularization, which also provides insights into how repetitive usage of data can lead to overfitting.

The regularized objective function, denoted by $\tilde{J}$, can be expressed as:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}) \tag{4.12}$$

Here, $(\boldsymbol{X}, \boldsymbol{y})$ represents the training set, $\boldsymbol{\theta}$ represents all the parameters including $\boldsymbol{w}$ and any unregularized parameters, J is the original objective function on the training data, $\Omega(\boldsymbol{\theta})$ is the parameter norm penalty, and α is a hyperparameter controlling the strength of the parameter norm penalty.

We can achieve weight decay regularization by setting $\boldsymbol{\theta} = \boldsymbol{w}$ and $\Omega(\boldsymbol{\theta}) = \frac{\alpha}{2}\|\boldsymbol{w}\|_2^2$, which yields the following equation:

$$\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) + \frac{\alpha}{2}\|\boldsymbol{w}\|_2^2 \tag{4.13}$$

To obtain the gradient of $\tilde{J}$ with respect to $\boldsymbol{w}$, we take the derivative of Eq.(4.13) with respect to $\boldsymbol{w}$, which yields:

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) + \alpha\boldsymbol{w} \tag{4.14}$$

To obtain the quadratic approximation of $\tilde{J}$, we use a Taylor series expansion around the minimum point $\boldsymbol{w}^*$:

$$\tilde{J}(\boldsymbol{w}) \approx \tilde{J}(\boldsymbol{w}^) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w})^\top \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*) \tag{4.15}$$

Here, $\boldsymbol{H}$ is the Hessian matrix of J with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$.

Gradient descent can be used to update the parameters as follows:

$$\boldsymbol{w}^{(\tau)} = \boldsymbol{w}^{(\tau-1)} - \epsilon\nabla_{\boldsymbol{w}}\hat{J}\left(\boldsymbol{w}^{(\tau-1)}\right) \tag{4.16}$$

$$= \boldsymbol{w}^{(\tau-1)} - \epsilon\boldsymbol{H}\left(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*\right) \tag{4.17}$$

where ϵ is the learning rate.

The value of α can be approximately calculated from Eq.(7.33)-(7.45) [42]:

$$\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* = (\boldsymbol{I} - \epsilon\boldsymbol{H})\left(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*\right) \tag{4.18}$$

Rewrite this expression by eigendecomposition $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$, where $\boldsymbol{\Lambda}$ is a diagonal matrix and $\boldsymbol{Q}$ is an orthonormal basis of eigenvectors

$$\begin{gathered}\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* = \left(\boldsymbol{I} - \epsilon\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\right)\left(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*\right) \\ \boldsymbol{Q}^\top\left(\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^*\right) = (\boldsymbol{I} - \epsilon\boldsymbol{\Lambda})\boldsymbol{Q}^\top\left(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*\right)\end{gathered} \tag{4.19}$$

Further,

$$(\boldsymbol{I} - \epsilon \boldsymbol{\Lambda})^{\tau} = (\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1} \alpha \tag{4.20}$$

We can derive the relationship between how much to apply weight decay and repetitive steps:

$$\alpha \approx \frac{1}{\tau \epsilon} \tag{4.21}$$

The study presented in [40] suggests that an increased number of training iterations (τ) leads to a decreased value of α, indicating less regularization during model training when we reuse data. This implies that the neural network is more likely to become overfitted. Additionally, even for a single training example, subjecting it to repeated training through data reuse with an increased τ has a greater risk of overfitting. Another study [114] investigates the manipulation of the order of SGD, which degrades the convergence performance of the model.

To mitigate overfitting, we can leverage the theoretical insights from the above analysis and reduce the repetitive times of a single training sample by decreasing the number of training iterations (τ). One way to achieve this is by increasing the shuffle buffer size and applying different data augmentation techniques to introduce more diversity into the training data. Additionally, adjusting hyperparameters, such as the learning rate, can improve the accuracy during the fine-tuning stage.

By reducing the number of training iterations, we can decrease the risk of overfitting and improve the generalization performance of the model. Therefore, it is crucial to carefully consider and tune the hyperparameters, as well as apply appropriate data preprocessing and augmentation techniques, in order to achieve optimal results. In the following sections, we will delve into these factors in greater detail, exploring their impact on model performance and discussing various approaches for mitigating their effects.

4.4 Reusing data in pre-training and fine-tuning for target accuracy

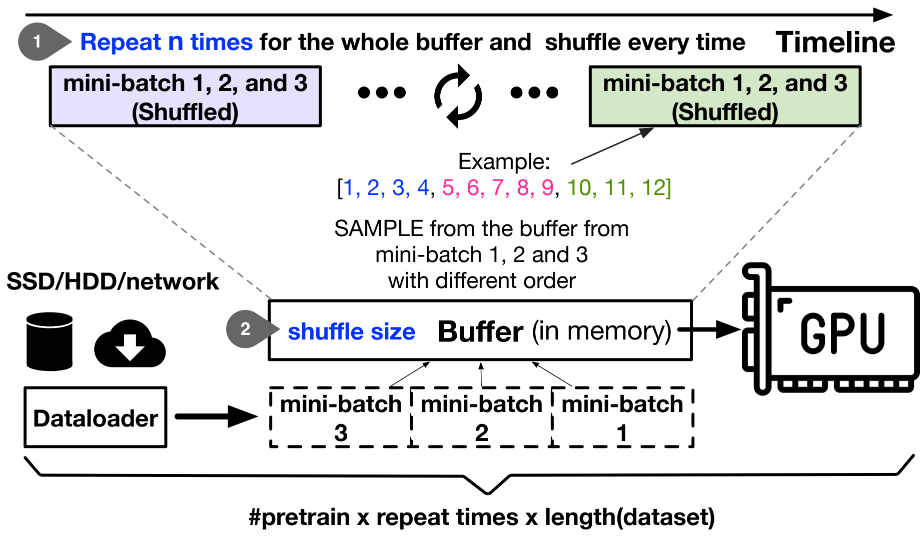
Pre-training and fine-tuning are used to improve the performance of a model on a specific task. We will delve into the details of how to pre-train and fine-tune a model to ensure the validation accuracy, including a discussion of the various approaches and strategies that can be used empirically.

Data echoing [29] and refurbishing partial data [66] are two methods that aim to improve the throughput of the data loader by reusing all or part of the preprocessing pipeline. These methods have been shown to ensure the convergence of the training process [8]. However, simply reusing the data without adjusting hyperparameters can be difficult to achieve the original optimal validation accuracy. Additionally, conducting a thorough hyperparameter search from scratch for a large model can be time-consuming.

4.4.1 Metrics to switch to fine-tuning.

Determining the optimal time to start fine-tuning after pre-training is critical for maximizing computation time and improving model accuracy. Fine-tuning allows the model to specialize in the target task by leveraging the knowledge acquired from reused data during pre-training. However, starting too early or too late can negatively impact the training efficiency and model's performance. Two metrics can be used to determine the optimal time to start fine-tuning: accuracy and loss value.

One effective approach is to monitor the model's accuracy during pre-training. The pre-training process should continue until the model reaches a plateau in accuracy, indicating that it has learned all it can from the available data. Once the model has reached this point, the fine-tuning phase can commence. During fine-tuning, we can experiment with a range of epochs to find the optimal number for fine-tuning while adjusting hyperparameters as needed. Another metric to consider is the loss value. Monitoring the loss value during pre-training and looking for a decrease in



1
Repeat n times for the whole buffer and shuffle every time
Timeline
mini-batch 1, 2, and 3 (Shuffled)
mini-batch 1, 2, and 3 (Shuffled)
Example:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
SAMPLE from the buffer from mini-batch 1, 2 and 3 with different order
SSD/HDD/network
2
shuffle size
Buffer (in memory)
GPU
Dataloader
mini-batch 3
mini-batch 2
mini-batch 1
#pretrain x repeat times x length(dataset)

Figure 4-8: Illustration of data loading with a shuffle buffer architecture, caching multiple mini-batches in memory and shuffling them for randomness. Data is loaded from SSD/HDD/network and transferred to GPU for model training.

the rate of change of the loss function can indicate that the model has reached a satisfactory pre-trained state and is ready for fine-tuning. However, it's worth noting that the optimal time for this transition may vary based on the dataset and task.

Besides, if time or computation budget is limited, it is better to allow the model to train for the maximum number of epochs of the original training process and then switch to fine-tuning at the end of the final fine-tuning stage, e.g., 1/4 of the total original training budget, for improving the accuracy.

Overall, determining the optimal time to start fine-tuning involves monitoring both accuracy and loss value during pre-training when they are stable and adjusting hyperparameters as needed during the fine-tuning phase. By doing so, we can optimize computation time and improve the model's accuracy for the target task. Additionally, it avoids starting hyperparameter search from scratch at the beginning of training.

4.4.2 Examining the Effect of Shuffle Buffer Size on Pre-training

Our study aims to examine how the size of a shuffle buffer affects the accuracy performance of a model when reusing mini-batches. The buffer size plays a crucial role in optimizing the data loading process for efficient model training. Firstly, it impacts the randomness of the generated mini-batches, which can affect the convergence of optimization algorithms such as stochastic gradient descent (SGD) [31]. Secondly, caching data without repetitive decoding can improve data preprocessing throughput as shown in Figure 4-3 but significantly increase the size of the decoded dataset by 5 - 7x times in memory [76]. Striking a balance between caching data for saving memory and maintaining sufficient randomness in the mini-batches is crucial for optimal model performance.

In addition, using an internal buffer, as depicted in Figure 4-8, can greatly enhance the performance and efficiency of the data loading process for training deep learning models. These buffers manage the loading and caching of pre-processed or unprocessed input data, creating a seamless pipeline for data flow. They can even prefetch data elements from the input dataset using background threads, anticipating the data needs of the downstream system and pre-loading data ahead of time to reduce waiting times.

To gain a deeper understanding of the correlation between shuffle size and training performance, we conduct experiments using shuffle sizes of 1 batch size (2048 training examples, 4.096% of the entire dataset), 2 batch sizes (8.192%), and 3 batch sizes (12.288%) on 4 GPUs, each repeated three times, with the CIFAR10 dataset and the ResNet18 model. By systematically altering the shuffle size and repeating the tests, we aim to gain insights into how varying shuffle sizes affect the randomness of mini-batches and the accuracy of the model training process.

Figure 4-9 demonstrates the impact of the shuffle buffer on training performance using the CIFAR10 dataset and a ResNet18 model. The results indicate that the accuracy of the model is significantly affected by different shuffle sizes. Among them, the 3 batch size condition performs the best, surpassing the baseline accuracy. The experiments utilized SGD optimizer, a learning rate of 0.1, momentum of 0.9, and weight decay of 5e-4. Data augmentation was applied. The baseline condition without the shuffle buffer achieves 93.25% accuracy. However, when the shuffle size is set to 1 batch size, the accuracy does not improve significantly and falls short of the target accuracy. The 2 batch size condition performs better, but still falls short of the target

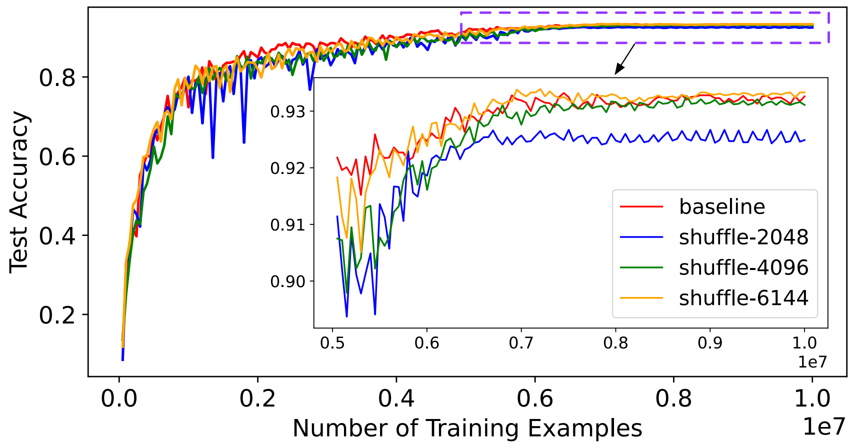


Figure 4-9: The effect of the shuffle buffer size on the accuracy of a ResNet18 model trained on the CIFAR10 dataset.

accuracy. Only the 3 batch size condition, which occupies 12.288% of the total dataset size, surpasses the baseline accuracy and reaches the desired target.

Findings of our experiment emphasize the significance of shuffle size in the pre-training phase's success. A larger shuffle size increases randomness, which reduces the τ value in Eq.(4.21), thus reducing the risk of overfitting and enhancing the model's performance, as demonstrated in the 3 batch size conditions' results. On the other hand, an inappropriately small shuffle size can prevent the model from reaching the target accuracy. Thus, choosing the shuffle size carefully is crucial when designing the data loading process for training a machine learning model.

But how can we still achieve the target accuracy when memory constraints limit the shuffle buffer size? Fine-tuning involves training a pre-trained model on a specific task by initializing it with pre-trained weights. The motivation behind fine-tuning is to leverage the features learned during pre-training on data with a similar distribution and adjust the model to the target dataset, ultimately achieving the desired validation accuracy. The primary issue of fine-tuning is that even though the data in pre-training may be reused, the ordering of the data is differ from the original order during pre-training, potentially affecting the behavior of SGD algorithms. In fact, [114] identified issues like batch-order poisoning and backdoors that can arise due to this discrepancy.

We conducted an experiment with a shuffle size of one batch (2048), which is

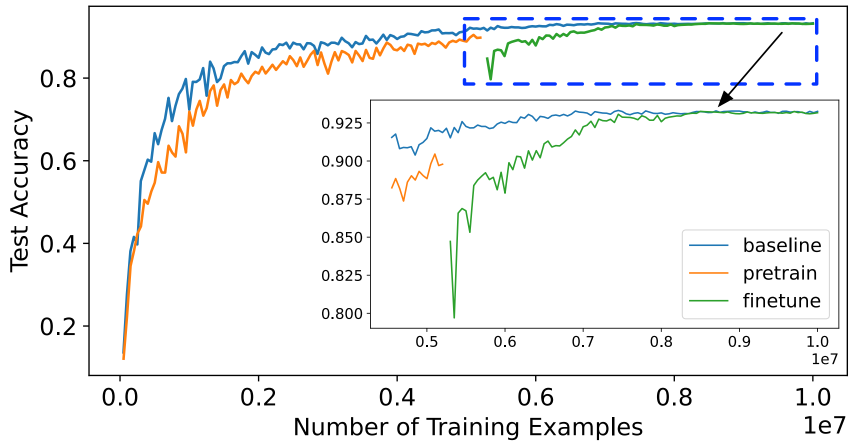


Figure 4-10: The effect of fine-tuning on the accuracy of a ResNet18 pre-trained for 35x3 epochs on the CIFAR10.

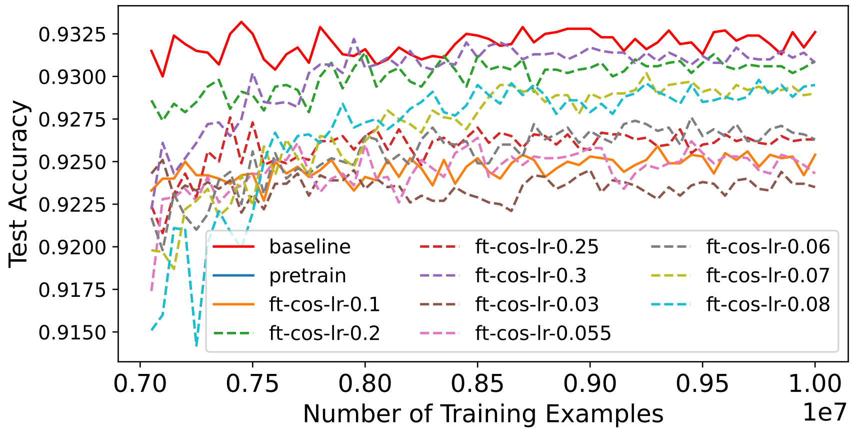


Figure 4-11: Accuracy of ResNet18 after 35x3 epochs of pretraining with unsuccessful learning rate scheduler and fine-tuning, shown for several trials.

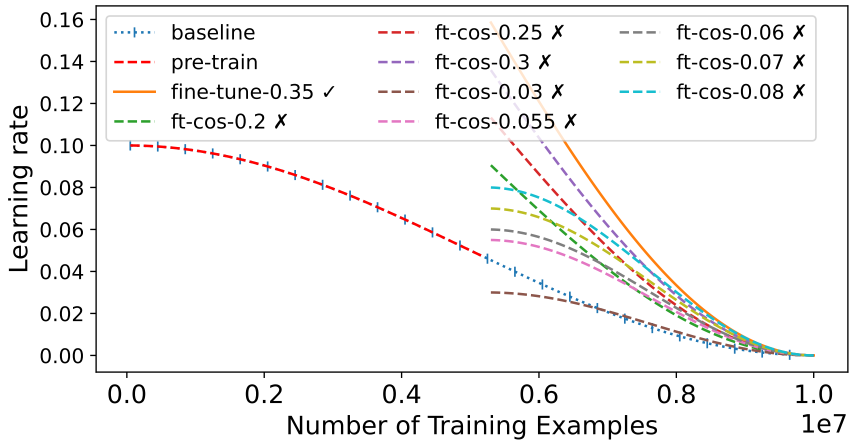


Figure 4-12: Learning rates for ResNet18 pre-trained on CIFAR10 with 35x3 epochs and shuffle size of 1, and then fine-tuned.

known to be insufficient for the ResNet18 model in Figure 4-9, and repeated it three times. The model underwent a pre-training phase of 35 epochs and was fine-tuned for an additional 95 epochs (from epoch 106 to 200 in the baseline condition). Our goal was to investigate the impact of fine-tuning on the accuracy of a ResNet18 model trained on the CIFAR10 dataset. We focused on the challenges posed by memory limitations and suboptimal data shuffling. Our results, depicted in Figure 4-10, demonstrate that fine-tuning significantly improved the accuracy of the model, reaching the baseline accuracy despite the memory limitations and inefficient shuffling. This supports the conclusion that fine-tuning can be an effective method for enhancing the performance of deep learning models, even under memory constraints.

The importance of selecting an appropriate learning rate for a model is widely recognized, as it can significantly affect performance. A learning rate scheduler allows for adjustment of the learning rate during training, in response to the current state of the model. In our study, we aimed to ensure that the learning rate was appropriately high during the initial stages of fine-tuning, to compensate for any differences between the original case and the drifted case. As training progressed, the learning rate gradually decreased to facilitate convergence to the desired level of accuracy.

During the fine-tuning stage of our experiments, we performed a manual search to identify the optimal initial learning rate using a cosine annealing-based learning rate scheduler that gradually reduced the learning rate as training progressed. We started with an initial learning rate of 0.35 and reduced it to 0.16 at the 106th baseline epoch. Figure 4-12 shows both successful and unsuccessful learning rate schedulers, with the unattained accuracy displayed in Figure 4-11. Our experiment emphasizes the importance of selecting the learning rate carefully during the fine-tuning stage of model training. The results highlight the criticality of this process, and by utilizing a learning rate scheduler and conducting a manual search for the optimal initial value, we can significantly enhance the model's performance and obtain better results.

The results from Figure 4-13 indicate that pre-training the ResNet18 model for 45 epochs repeated 3 times with a shuffle size of 1 batch size failed to achieve the desired accuracy. Despite our attempts to fine-tune the model, the performance remained

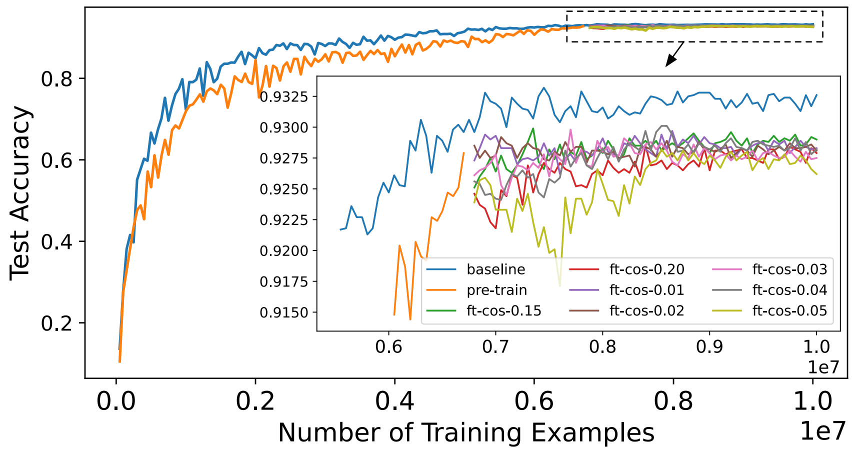


Figure 4-13: Accuracy of ResNet18 pre-trained for 45 epochs repeated 3 times with a shuffle buffer size of 1 batch, and fine-tuned from epochs 136 to 200.

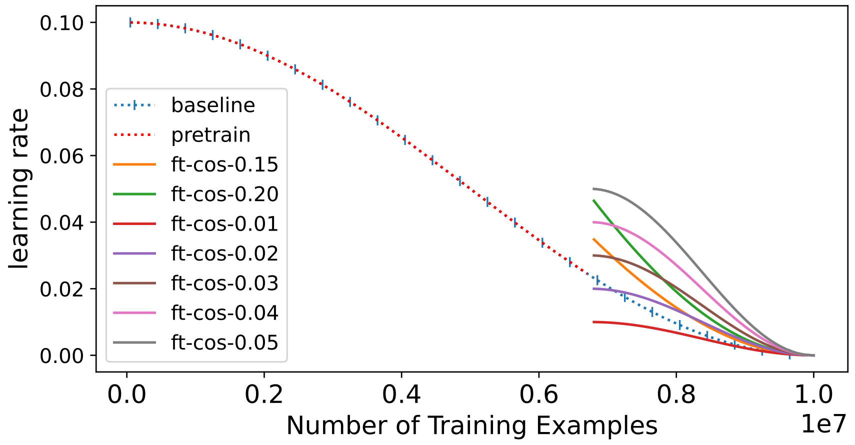


Figure 4-14: ResNet18 pre-trained for 45 epochs repeated 3 times, and fine-tuned from epochs 136 to 200 with various initial learning rates.

unsatisfactory, possibly due to the insufficient shuffle size, which may have limited the model's generalization ability. To address this issue, we increased the shuffle size to 2 batch sizes in Figure 4-15, which accounted for 8.2% of the data. This change led to the achievement of the target accuracy.

Figure 4-14 presents the learning rate curve for the ResNet18 model pre-trained for 45 epochs repeated 3 times (i.e., 135 epochs in total in pre-training). Fine-tuning was conducted from epochs 136 to 200 with different initial learning rates, as indicated in the legend. Despite our efforts to optimize the learning rate, the results were insufficient to reach the target accuracy. However, in Figure 4-15, we present the

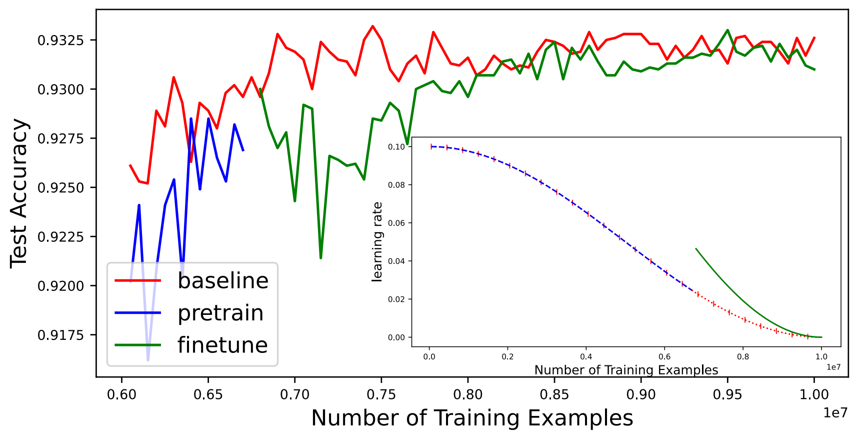


Figure 4-15: Fine-tuning of ResNet18 model with pre-training of 45 epochs repeated 3 times, using a shuffle buffer size of 2 batches.

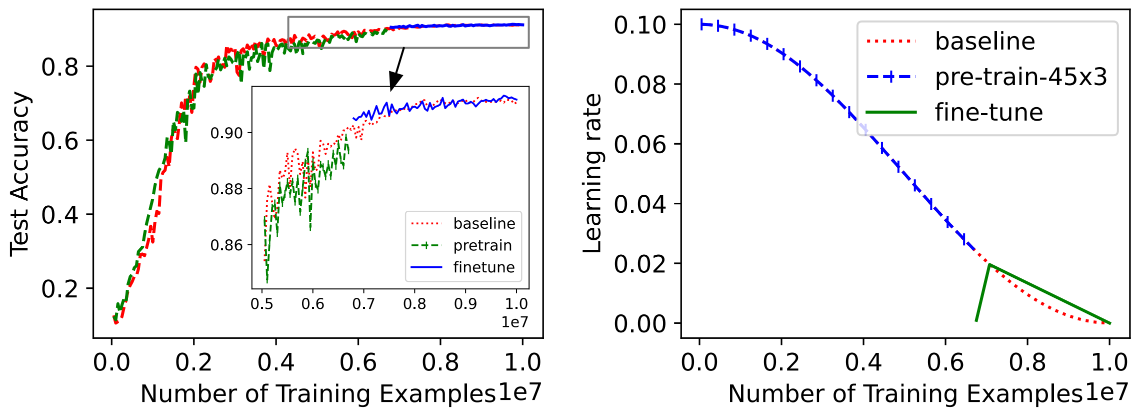


Figure 4-16: The VGG16 model was pre-trained for 45 epochs repeated 3 times. The accuracy and learning rate were plotted in the left and right figures, respectively.

learning curve for the model, which utilized the ResNet18 and was pre-trained for 45 epochs with a shuffle buffer size of 2 batch sizes, repeated 3 times. Fine-tuning was carried out from epochs 136 to 200 with an initial learning rate of 0.2 and a cosine annealing scheduler, which enabled us to easily reach the baseline accuracy. In addition, we demonstrate the training process of VGG16 architecture pre-trained for 45 epochs, repeated three times, using a different learning rate scheduler in Figure 4-16. We were able to achieve the desired level of accuracy by fine-tuning the model with data augmentation and a shuffled batch size of 2432. This is highlighted in our results.

We conducted an experiment to validate our findings on a larger dataset using the

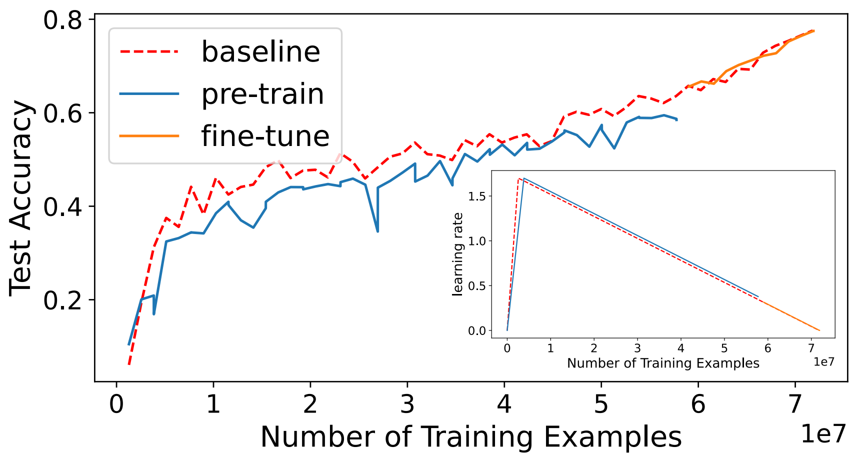


Figure 4-17: Accuracy and learning rate curves for a ResNet50 model trained on Imagenet dataset with 56 epochs (15 epochs of pre-training repeated 3 times and 11 epochs of fine-tuning).

Imagenet-1K dataset [33] using the FFCV data loader library [64]. The experiment involved training a ResNet50 model for 15 epochs of pre-training, which was repeated three times, followed by 10 epochs of fine-tuning, resulting in a total of 55 training epochs. We use the OneCycle learning rate scheduler and the learning rate was initialized at a peak of 1.7 at the first epoch, with a batch size of 512 per GPU and four GPUs in total. We used SGD optimization with a momentum of 0.9, weight decay of 1e-4, and label smoothing of 0.1. The results are shown in Figure 4-17, where the accuracy improved during fine-tuning and reached the original baseline accuracy of 77.5%. Prior to fine-tuning, the accuracy was lower than the baseline. To optimize the training process, we set the shuffle buffer size to 8192, which is equal to 16 times the batch size of 512, representing 0.64% of the entire Imagenet-1K dataset.

4.4.3 Without augmentation in pre-training

We have chosen not to use data augmentation in our approach to further conserve computational resources during data preprocessing. By foregoing operations such as rotation, cropping, and scaling, we can minimize the time and memory required. However, this approach may result in a dataset that is biased towards less augmented versions of the original examples, potentially impeding the model's capacity to gen-

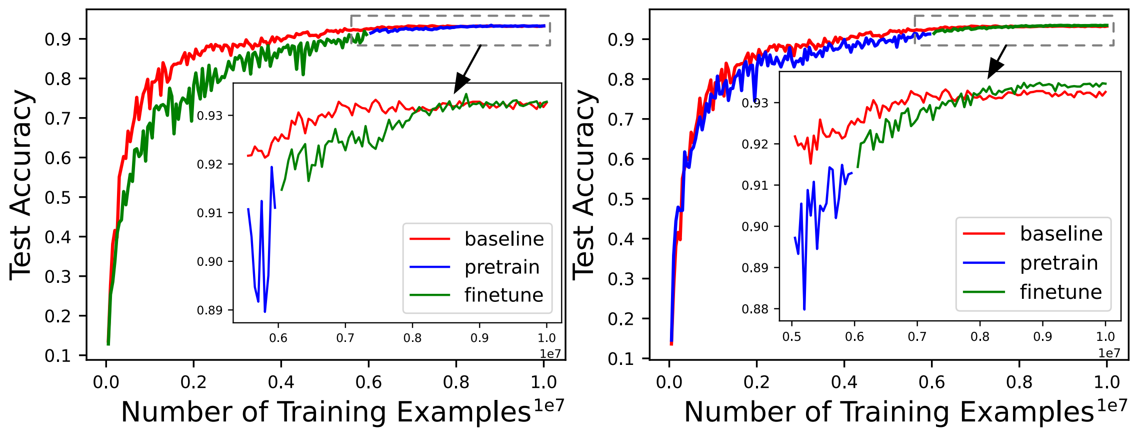


Figure 4-18: Accuracy of ResNet18 models pretrained for 40x3 epochs each (120 epochs total). The left figure used shuffled batch size of 1 with data augmentation, while the right figure had immediate reuse without augmentation. Both models reached the target accuracy after fine-tuning from epochs 121 to 200.

eralize to new data.

Data augmentation is a technique used to increase the amount of new and synthetic data examples available for training a model. By expanding the size of the training dataset, data augmentation reduces the number of times a single example is utilized during the training process, i.e., reduce the τ in Eq.(4.21). This helps to mitigate the risk of overfitting, where a model memorizes the training data rather than learning the underlying patterns. By reducing the number of repeated exposures to a single example, data augmentation prevents overfitting and results in more robust and accurate models.

To address overfitting and achieve optimal accuracy, we incorporate the augmented data into the fine-tuning process after a certain number of pre-training epochs. This allows the model to learn from the original examples with increased augmentation transformations. This strategy balances the trade-off between computational efficiency and generalization performance. It is essential to consider both the advantages and disadvantages of data augmentation while making this decision.

In this experiment, we aim to demonstrate that immediate data reuse without augmentation can be a practical option in certain scenarios, particularly after fine-tuning. We evaluated the performance of ResNet18 models pre-trained for a total

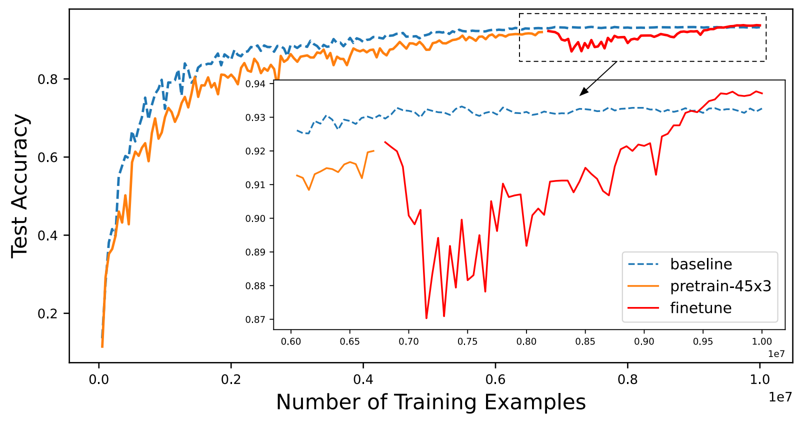


Figure 4-19: Accuracy of ResNet18 on CIFAR-10 using immediate reuse without augmentation, with 3 repetitions and OneCycle learning rate scheduler during fine-tuning.

of 120 epochs by repeating the training example for three times for 40 epochs in the pre-training process. The accuracy results are presented in Figure 4-18. The left graph illustrates the outcomes when the shuffle size was set to 1 batch, and data augmentation was applied each time the data was reused. In contrast, the right graph shows the results when immediate reuse was performed without augmentation. We further conducted fine-tuning from epoch 121 to 200. The findings indicate that both models achieved the desired accuracy after fine-tuning. These results suggest that immediate data reuse without data augmentation can be a viable option in certain cases, especially after fine-tuning. However, it is essential to note that without data augmentation, the fine-tuning process becomes more challenging to reach the desired accuracy. Therefore, a careful consideration of the trade-offs between computational efficiency and impact on model generalization must be made before deciding whether to use data augmentation. We will discuss these considerations further in the following sections.

Enhancing Model Performance through Fine-Tuning after Immediate Data Reuse. Our objective is to achieve a target accuracy by fine-tuning the model. Our initial experiments revealed that immediate data reuse without augmentation did not consistently achieve the target accuracy on the CIFAR-10 dataset using ResNet18 models. Thus, we explored the use of fine-tuning during the final training phase as a

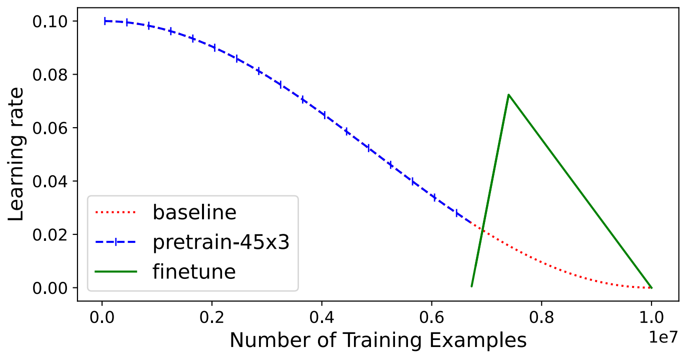


Figure 4-20: OneCycle learning rate schedule during fine-tuning for a ResNet18 model on CIFAR-10 using immediate reuse without augmentation and 3 repetitions.

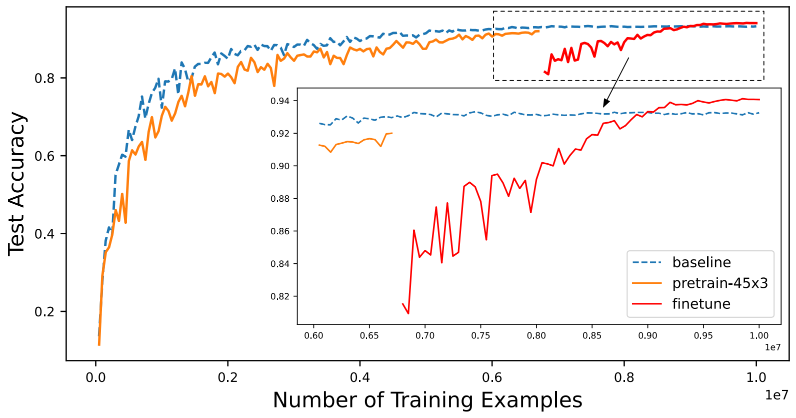


Figure 4-21: Accuracy of ResNet18 on CIFAR-10 using immediate reuse with 3 repetitions and cosine annealing learning rate scheduler during fine-tuning.

potential solution to improve model performance.

Figure 4-19 depicts the accuracy of ResNet18 models on the CIFAR-10 dataset using immediate reuse without augmentation and with fine-tuning. The models were pre-trained for 45 epochs, resulting in a total of 135 epochs, followed by fine-tuning from epoch 136 to 200 using OneCycleLR with a peak learning rate of 0.0723, weight decay of 0.0005, batch size of 896, and momentum of 0.9. The hyper-parameters were selected using a Weights&Biases Sweeps [16]. We were able to achieve the target accuracy during the fine-tuning phase. Figure 4-20 displays the learning rate schedule for OneCycleLR with a peak learning rate of 0.0723. Our findings suggest that fine-tuning can effectively enhance the performance of models using immediate data reuse without augmentation.

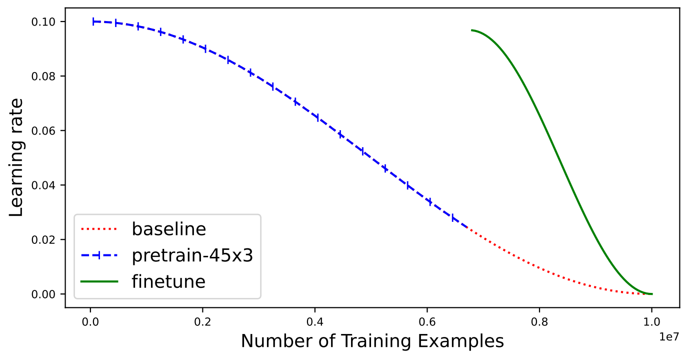


Figure 4-22: Cosine annealing learning rate schedule used during pre-training and fine-tuning for ResNet18 on CIFAR-10 dataset.

Impact of Learning Rate Scheduler Strategies on Fine-Tuning. We evaluated the impact of two learning rate scheduler strategies on the accuracy of deep learning models during fine-tuning. Our experiments demonstrated that both the One Cycle policy and cosine annealing learning rate schedulers improved model accuracy, as depicted in Figures 4-20 (One Cycle) and 4-22 (Cosine Annealing). In our experiments, we used a shuffle size of 3 times the batch size, a total of 200 epochs, and 3 repetitions of 45 pre-training epochs using ResNet18 for both learning rate scheduler strategies. The accuracy results of the models are presented in Figure 4-21. Our findings indicate that utilizing various learning rate schedulers can significantly improve model accuracy during fine-tuning, provided that we identify appropriate hyperparameters.

To verify that our findings are not limited to a specific type of model, we conducted an additional experiment using the MobileNetV2 architecture. Specifically, we pre-trained the MobileNetV2 model on the CIFAR-10 dataset for 45 epochs with immediate data reuse, which was repeated three times. We then fine-tuned the model for 65 epochs without data augmentation and evaluated its performance. The results of this experiment are shown in Figure 4-23. MobileNetV2 can achieve the desired level of accuracy.

Immediate Data Reuse Results on ResNet50 for ImageNet. We conducted an experiment to evaluate the effectiveness of immediate data reuse on ResNet50 for the large-scale ImageNet dataset. Our results are presented in Figure 4-24. In this experiment, we aimed to achieve the desired level of accuracy by using immediate

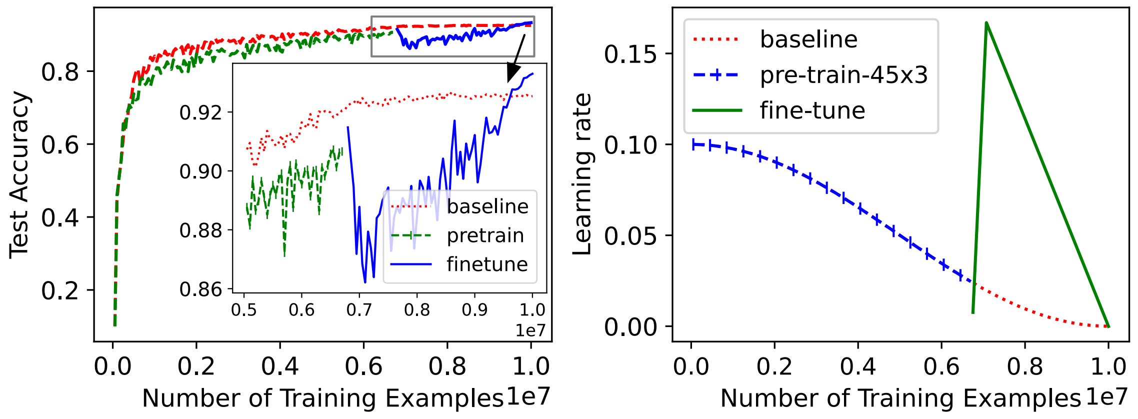


Figure 4-23: Accuracy and learning rate of MobileNetV2 trained on the Cifar10 using immediate data reuse. The model was pre-trained for 45 epochs repeated 3 times and fine-tuned 65 epochs.

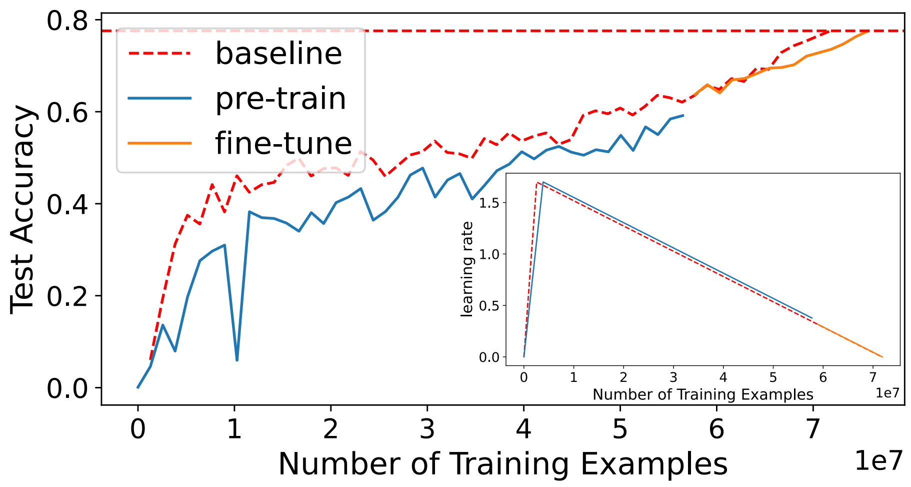


Figure 4-24: Accuracy of ResNet50 trained on the ImageNet dataset using immediate data reuse. The model was trained for 60 epochs, consisting of 45 pre-training epochs (repeated 3 times) and 15 fine-tuning epochs.

data reuse. The baseline training lasted for 56 epochs, and the immediate data reuse case involved 45 pre-training epochs (repeated three times) and 15 fine-tuning epochs, totaling 60 epochs, which is four more than the baseline training. Our results demonstrate that immediate data reuse improved the training process and enabled us to achieve the desired level of accuracy.

Our study showed that reusing data without applying augmentation techniques can be more challenging than reusing data while using different augmentation methods

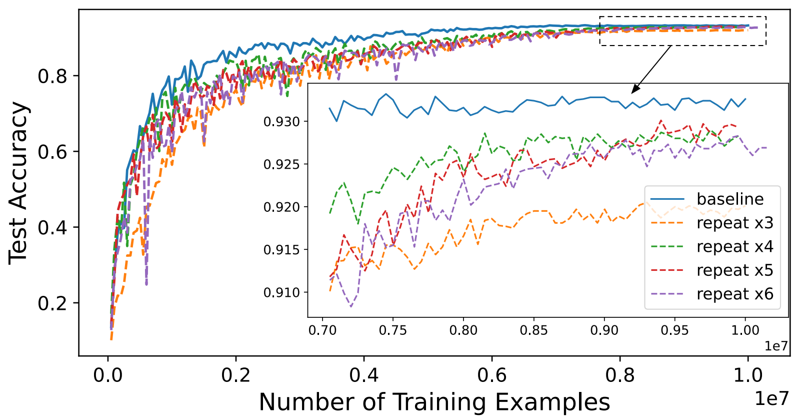


Figure 4-25: Accuracy of ResNet18 on the CIFAR-10 using immediate reuse without augmentation, with varying numbers of repetitions. The models were trained for 200 epochs, but were unable to reach the target accuracy.

to increase the size of the dataset, as shown in Figure 4-17. Although our immediate data reuse approach required an additional four epochs to achieve the target accuracy, it was accomplished at an acceptable cost. These findings suggest that immediate data reuse may require more effort during fine-tuning to attain the desired accuracy compared to reusing data with various augmentation techniques. Our results indicate that using data augmentation techniques during data reuse can help achieve the desired accuracy more efficiently than immediate data reuse alone.

4.4.4 The Effect of Repeat Times on Fine-tuning

Pre-training with Varying Data Reuse Frequencies. In this study, we aimed to investigate the impact of data reuse frequency on model performance during pre-training. We conducted experiments with different numbers of repetitions to immediately reuse the same data, but the target accuracy was not achieved, possibly due to oversampling [22]. Specifically, we pre-trained models with repetition counts of 3, 4, 5, and 6, using a shuffle buffer size of 1 batch size. However, as shown in Figure 4-25, these models failed to attain the desired accuracy.

Repeated use of the same data during pre-training, as shown in Eq.(4.21), can lead to overfitting, making it harder to achieve the desired accuracy during fine-tuning. Thus, understanding the impact of data reuse frequency on model performance is

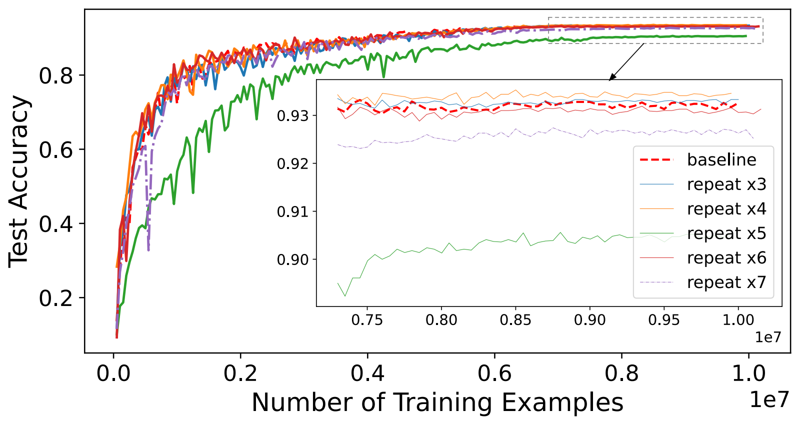


Figure 4-26: Accuracy of ResNet18 with varying repeat times during pre-training with shuffle size as 3 times of batch size (2048) and different data augmentation.

critical in designing effective pre-training processes.

As depicted in Figure 4-26, our experiments have highlighted a significant relationship between data reuse frequency and shuffle size. Specifically, increasing the number of times data is repeated may lead to a decrease in model accuracy, likely because of overfitting. This occurs because repeatedly using the same data can cause the model to become too focused on specific details, making it perform poorly on new and unseen data. Conversely, increasing the shuffle size can improve accuracy by introducing a more diverse range of data. This diversity can help prevent overfitting and enhance the model's ability to generalize to new data compared with the Figure 4-25. Therefore, determining the optimal data reuse frequency and shuffle size is crucial for achieving the best pre-training performance.

To summarize, our findings suggest that while immediate data reuse can be helpful to save memory and reduce computation overhead, it's essential to balance it with the introduction of new and diverse data to prevent overfitting. This can be achieved by increasing the shuffle size to introduce more varied data into the training process.

As part of our investigation, we aimed to assess the impact of repeating a large amount of data during pre-training on the accuracy and learning rate of the ResNet18 model. Specifically, we wanted to determine whether this approach could still lead to improved model accuracy during fine-tuning, even with a high number of data repeats. To do this, we trained the ResNet18 model for a total of 200 epochs, with 20

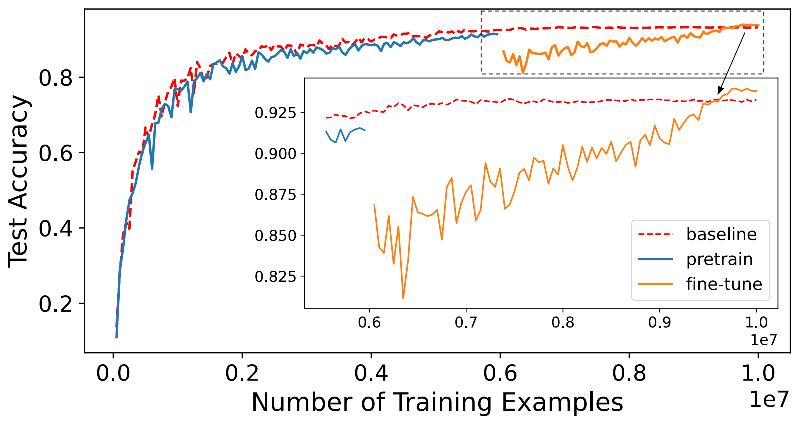


Figure 4-27: Accuracy of ResNet18 improved with a 3x batch size shuffle, repeated 6 times over 200 epochs, through pretraining for 20x6 epochs and fine-tuning for 80 epochs.

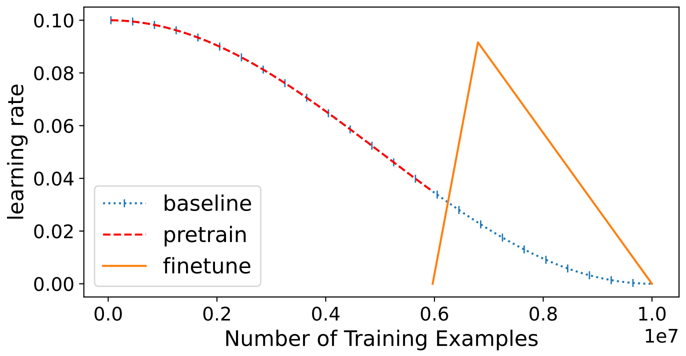


Figure 4-28: Learning rate in the fine-tuning stage with a 3x batch size shuffle buffer, repeating 6 times over 200 epochs using ResNet18.

epochs dedicated to pre-training and repeated 6 times, and 80 epochs for fine-tuning. During training, we employed data augmentation and set the shuffle size to three times the batch size to enhance the model's ability to generalize.

Our results, as illustrated in Figure 4-27, demonstrated that repeating a large amount of data during pre-training can indeed lead to improved model accuracy during fine-tuning. The model was able to achieve the desired accuracy compared to the baseline, even with a high number of data repeats.

However, it is important to note that we observed significant changes in the learning rate scheduler during both pre-training and fine-tuning, as shown in Figure 4-28. This suggests that the learning rate needs to be heavily tweaked when employing this

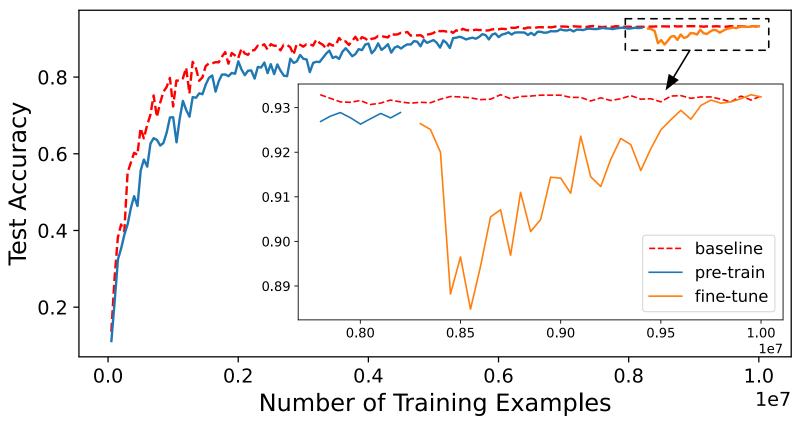


Figure 4-29: Accuracy of ResNet18 pre-training for 55 epochs with 3 repetitions, and fine-tuned for 35 epochs.

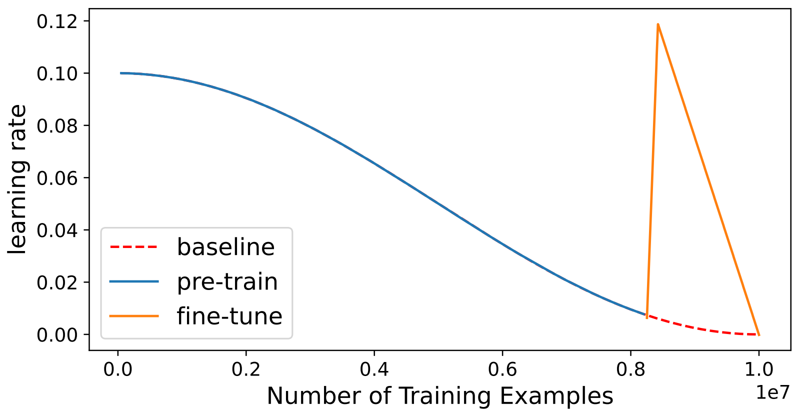


Figure 4-30: Learning rate of ResNet18 pre-training for 55 epochs with 3 repetitions, and fine-tuned for 35 epochs.

approach. Therefore, while repeating a large amount of data during pre-training can be an effective way to enhance model accuracy during fine-tuning, it is important to carefully consider the learning rate changes that may occur as a result.

4.4.5 The Timing to Fine-tuning

Analysis of Pre-Training Epochs for Immediate Data Reuse. To investigate the effectiveness of pre-training for immediate data reuse, we conducted experiments with pre-training epochs of 45x3, 55x3, and 65x3. Analyzing the results of different fine-tuning start epochs enables us to determine the trend of timing for fine-tuning after pre-training. It is worth noting that fine-tuning for immediate data reuse, where

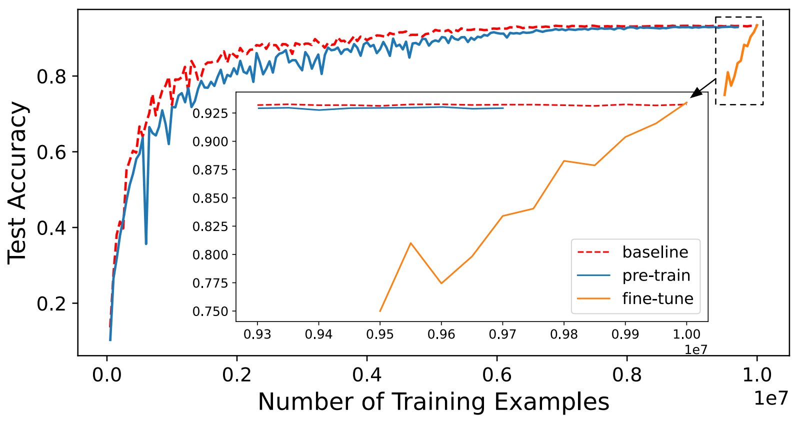


Figure 4-31: Accuracy of ResNet18 with 3 repetitions, pre-training for 65 epochs, and fine-tuned for 10 epochs.

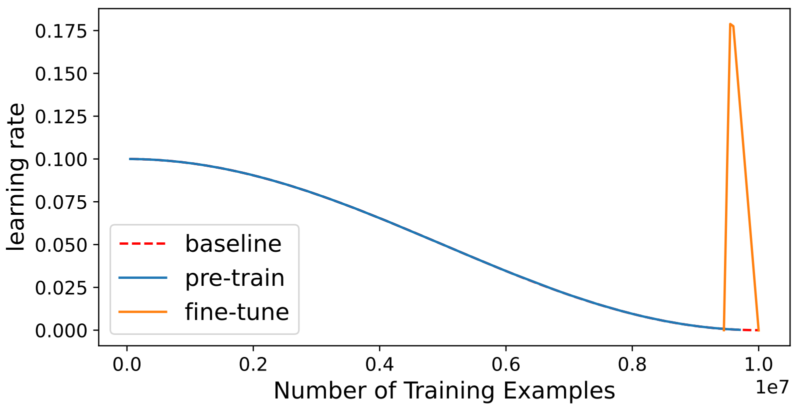


Figure 4-32: Learning rate of ResNet18 with 3 repetitions, pre-training for 65 epochs, and fine-tuned for 10 epochs.

the pre-trained model is used without any data augmentation, is more challenging than fine-tuning for data reuse after augmentation. This is because fine-tuning requires a higher learning rate to compensate for the discrepancy between the baseline and the model based on the repetitive data distribution. In the previous section, we discussed the results of pre-training at 45x3 epochs, as shown in Figure 4-19. Here, we further evaluate the effectiveness of pre-training at different epochs and their impact on fine-tuning performance.

Fine-Tuning for Immediate Reuse Case: Pre-train 55x3, Total 200 Epochs. In our experiments, we pre-trained the ResNet18 model with a batch size of 2048 for 4 GPUs, weight decay of 0.0005, and momentum of 0.9, repeating the process 3 times

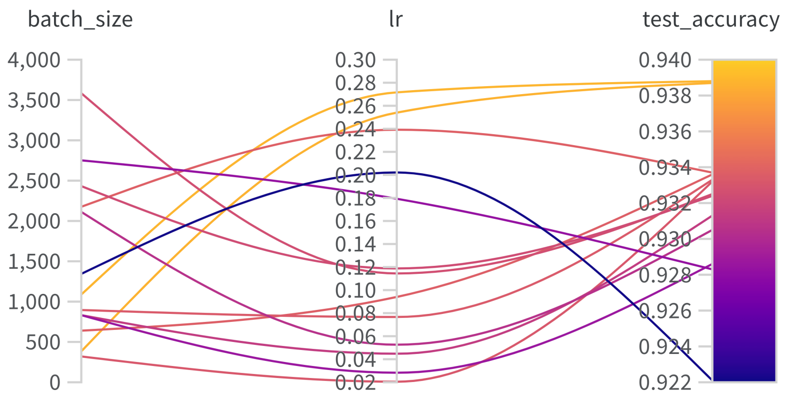


Figure 4-33: W&B Sweep hyperparameter searching for improved performance of resNet18 model with 3 Repetitions and pre-training for 55 epochs.

for a total of 55x3 epochs. Then, we fine-tuned the model for 35 epochs with a batch size of 2432 using a OneCycle learning rate scheduler with a peak learning rate of 0.118, as depicted in Figure 4-30. The results showed that the model achieved an accuracy of 93.43%, as depicted in Figure 4-29. To further evaluate the model's performance, we conducted a hyperparameter search, as depicted in Figure 4-33. In the hyperparameter searching, we select two main hyperparameters to tune the test accuracy. The Sweep results indicated that the combination of 55x3 epochs of pretraining and 35 epochs of fine-tuning was the most effective in reaching the target accuracy.

Fine-Tuning for Immediate Reuse Cases: Repeat Pretraining 65x3, Total 205 Epochs. In our experiments, we first pretrained the ResNet18 model using a batch size of 2048 and 4 GPUs, with a weight decay of 0.0005 and a momentum of 0.9. We repeated this process 3 times, for a total of 65x3 epochs. Then, we fine-tuned the model with a batch size of 320 for an additional 10 epochs, using a OneCycleLR learning rate scheduler with a peak learning rate of 0.195, as shown in Figure 4-32. Our results indicated that the model achieved an accuracy of 93.43%, as shown in Figure 4-31. We also performed a hyperparameter sweep to achieve the desired accuracy.

Our findings suggest that it's possible to achieve high accuracy with short fine-tuning periods, even after extensive pre-training. However, it's important to balance computational efficiency with model generalization when determining the extent of pre-training for efficient fine-tuning. For fine-tuning starting from 65x3 epochs, a higher peak learning rate is required compared to the 55x3 case, as the former has fewer opportunities to converge to the optimal valley. A higher learning rate compensates for the drifted hyperplane distance in high-dimensional space.

4.4.6 Discuss about learning rate scheduler and hyperparameters

When fine-tuning a pre-trained model for data reuse, selecting the appropriate hyperparameters is critical. Several key factors must be taken into account, including the initial learning rate and learning rate scheduler, batch size, and various optimizer hyperparameters such as weight decay coefficient. Choosing the right hyperparameters requires careful consideration of the specific task and an understanding of the various trade-offs involved. To optimize hyperparameters effectively, it is essential to experiment with different combinations of hyperparameters and optimizer settings. This approach allows for the identification of the best possible solution for a particular task.

One effective way to explore different hyperparameters is to use hyperparameter tuning techniques, such as those offered by Sweeps [5]. These techniques can help to automate the search process and make it more efficient. Therefore, selecting the right hyperparameters is crucial when fine-tuning a pre-trained model for data reuse. Careful consideration of the specific task and experimentation with different hyperparameter and optimizer combinations are essential. Using hyperparameter tuning techniques can help to streamline this process and improve the overall efficiency of the search.

When fine-tuning a pre-trained model for data reuse, it is essential to consider several trade-offs in order to achieve the desired accuracy. One trade-off that can be made is selecting several key hyperparameters to optimize during fine-tuning. These hyperparameters include the initial learning rate and batch size. By focusing on these key hyperparameters, we can reduce the number of trials needed to achieve optimal results. In addition to hyperparameters, the choice of optimizer [79] and its associated hyperparameters can significantly impact the results of fine-tuning. Different optimizers may be more or less suitable for specific types of problems and datasets. It is important to carefully evaluate the performance of different optimizers and their hyperparameters to determine the best solution for a particular task.

Therefore, selecting the right hyperparameters and optimizer for fine-tuning a pre-trained model for data reuse is essential in achieving the desired accuracy. Careful consideration of the specific task, as well as the trade-offs involved, is necessary.

4.5 Related Work

Data Reuse. [29] introduces *data echoing* to reduce the computation used by earlier stages of the training pipeline and speed up neural network training. By reusing intermediate outputs from earlier pipeline stages, the proposed technique can match the baseline's predictive performance using less upstream computation. [66] proposes *data refurbishing*, a sample reuse mechanism that accelerates training by reusing partially augmented samples to reduce CPU computation while preserving sample diversity obtained by data augmentation. Both papers address the issue of reducing computation and accelerating the training process of deep neural networks by reusing data and features. However, *data echoing* requires hyperparameter tuning at the beginning of each training process to achieve optimal accuracy, which is time-consuming. On the other hand, *data refurbishing* still faces the challenge of not being able to reach the target accuracy.

Feature Reuse. Traditional backpropagation algorithm for training neural networks requires sequential processing, which limits parallelization and computing efficiency. [56] propose a novel parallel-objective formulation and features replay algorithm to address the limitations of sequential processing in backpropagation for training deep neural networks, achieving faster convergence, lower memory consumption, and better generalization error compared to other methods. This approach is more general, as it can reuse hidden feature data as well.

Convergence, Accuracy, and Imbalanced Data. Agarwal et al. [8] examine the convergence of data-echoed extensions of standard optimization methods, which builds upon the work proposed by Choi et al. [29]. However, the paper does not delve into achieving desired accuracy or fine-tuning hyperparameters. Additionally, we have identified an issue with Theorem 7 in the paper and have reached out to the author to clarify that the step size should be $\frac{D}{2K\rho}\sqrt{\frac{B}{T}}$ instead of $\frac{\rho}{KD}\sqrt{\frac{B}{T}}$.

Imbalanced datasets pose a challenge for deep learning algorithms [22]. Cao et al. propose methods such as re-weighting, re-sampling, cost-sensitive learning, margin-based loss functions, and generalization bounds to address this issue. Our findings suggest that fine-tuning using the original dataset can also help correct imbalanced and re-ordered datasets that affect optimization algorithms. Shumailov et al. [114] introduce training-time attacks that can disrupt model training or introduce backdoors in machine learning models by manipulating the order of data without changing the underlying dataset or model architecture. The authors observe that even a single adversarially-ordered epoch can slow down model learning or reset all learning progress. It is crucial to note that the order and distribution of data significantly affect stochastic optimization algorithms.

Dataloader as a Service. To mitigate data stalls, [76] utilizes a distributed in-memory cache in their data-loading library, CoorDL, as a simple but effective technique to speedup the data loading stage. In the meantime, the significance of comprehending data storage and ingestion for training large-scale deep recommendation models using datacenter-scale AI training clusters is emphasized in [136]. The paper introduces Meta's comprehensive a data storage and ingestion (DSI) pipeline, which includes a central data warehouse constructed on distributed storage and a Data PreProcessing Service that is scalable enough to eliminate data stalls. Additionally, the paper illustrates how diverse and continuous training jobs are used to collaboratively train hundreds of models across geo-distributed datacenters. Furthermore, Cachew [48] built for TensorFlow suggests autoscaling and autocaching policies to offer a comprehensive data loader service for data processing. These policies allow resources to scale dynamically to prevent training job stalls and automatically apply caching to reuse preprocessed data within and across jobs whenever it is performance/cost-effective.

4.6 Conclusion

In conclusion, the proposed *PerFect* method offers a new approach to optimize the data preprocessing stage in DL systems, with a two-phase training process consisting of pre-training and fine-tuning. This method not only enables the retention of desired accuracy targets but also reduces the overall training time by reusing cached data. Our experiments demonstrate that *PerFect* effectively achieves the desired accuracy while also decreasing training time. Our work provides theoretical insights into the trade-offs and limitations of this method, and offers practical guidance for practitioners to adopt and utilize *PerFect* in their own DL workflows.

Bibliography

[1] TensorFlow Lite: Deploy machine learning models on mobile and IoT devices. https://www.tensorflow.org/lite.

[2] How to prevent tensorflow from allocating the totality of a gpu memory, December 2015.

[3] English-german wmt'16 translation task, 2016.

[4] Tips to improve performance for popular deep learning frameworks on cpus, 2018.

[5] Tune hyperparameters, 2023.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A System for Large-scale Machine Learning. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[7] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. The Case for GPGPU Spatial Multitasking. In *In Prof. of IEEE IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2012.

[8] Naman Agarwal, Rohan Anil, Tomer Koren, Kunal Talwar, and Cyril Zhang. Stochastic optimization with laggard data pipelines. *arXiv preprint arXiv:2010.13639*, 2020.

[9] Zeeshan Ahmed, Saeed Amizadeh, Mikhail Bilenko, Rogan Carr, Wei-Sheng Chin, Yael Dekel, Xavier Dupre, Vadim Eksarevskiy, Senja Filipi, Tom Finley, et al. Machine learning at microsoft with ml. net. In *Proc. of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[10] David P Anderson. Boinc: A system for public-resource computing and storage. In *Fifth IEEE/ACM international workshop on grid computing*. IEEE, 2004.

[11] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys'22)*, 2022.

[12] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J Rossbach, and Onur Mutlu. Mask: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[13] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *arXiv preprint arXiv:1409.0473*, 2014.

[14] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[15] Simon Séhier Bert Hubert, Jacco Geul. Wonder shaper, September 2022.

[16] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[17] Rajarshi Biswas, Xiaoyi Lu, and Dhabaleswar K Panda. Designing a micro-benchmark suite to evaluate grpc for tensorflow: Early experiences. 2018.

[18] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188*, 2022.

[19] Alexander Borzunov, Max Ryabinin, Tim Dettmers, Quentin Lhoest, Lucile Saulnier, Michael Diskin, Yacine Jernite, and Thomas Wolf. Training transformers together, 2022.

[20] Thomas Bradley. Hyper-Q. http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf, 2013.

[21] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Proceedings of the Advances in neural information processing systems (NeurIPS'20)*, 2020.

[22] Kaidi Cao, Colin Wei, Adrien Gaidon, Nikos Arechiga, and Tengyu Ma. Learning imbalanced datasets with label-distribution-aware margin loss. *Advances in neural information processing systems*, 32, 2019.

[23] Richard Caruana. Multitask learning: A knowledge-based source of inductive bias. In *Proc. of the Tenth International Conference on Machine Learning*, 1993.

[24] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. EffiSha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU. In *Proc. of Symposium on Principles and Practice of Parallel Programming (SIGPLAN)*, 2017.

[25] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[26] Yang Cheng, Dan Li, Zhiyuan Guo, Binyao Jiang, Jiaxin Lin, Xi Fan, Jinkun Geng, Xinyi Yu, Wei Bai, Lei Qu, et al. Dlbooster: Boosting end-to-end deep learning workflows with offloading data preprocessing pipelines. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–11, 2019.

[27] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. In *arXiv preprint arXiv:1410.0759*, 2014.

[28] Dami Choi, Alexandre Passos, Christopher J Shallue, and George E Dahl. Faster Neural Network Training with Data Echoing. In *arXiv preprint arXiv:1907.05550*, 2019.

[29] Dami Choi, Alexandre Passos, Christopher J Shallue, and George E Dahl. Faster neural network training with data echoing. *arXiv preprint arXiv:1907.05550*, 2019.

[30] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Rachata Ausavarungnirun, Myeongjae Jeon, Youngjin Kwon, and Jeongseob Ahn. Memory harvesting in {Multi-GPU} systems with hierarchical unified virtual memory. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC'22)*, 2022.

[31] Jichan Chung, Kangwook Lee, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. Ubershuffle: Communication-efficient data shuffling for sgd via coding theory. *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[32] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A Low-latency Online Prediction Serving System. In *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[33] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[34] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *arXiv preprint arXiv:1810.04805*, 2018.

[35] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *arXiv preprint arXiv:1810.04805*, 2018.

[36] Michael Diskin, Alexey Bukhtiyarov, Max Ryabinin, Lucile Saulnier, Quentin Lhoest, Anton Sinitsin, Dmitry Popov, Dmitriy Pyrkin, Maxim Kashirin, Alexander Borzunov, Albert Villanova del Moral, Denis Mazur, Ilia Kobelev, Yacine Jernite, Thomas Wolf, and Gennady Pekhimenko. Distributed deep learning in open collaborations. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS'21)*, 2021.

[37] EleutherAI. Eleutherai: A grassroots collective of researchers working to open source ai research., 2022.

[38] Saar Eliad, Ido Hakimi, Alon De Jagger, Mark Silberstein, and Assaf Schuster. Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism. In *Proceedings of the USENIX Annual Technical Conference (ATC'21)*, 2021.

[39] Joaquin Anton Guirao etc. Fast ai data preprocessing with nvidia dali, November 2018.

[40] Benyamin Ghojogh and Mark Crowley. The theory behind overfitting, cross validation, regularization, bagging, and boosting: tutorial. *arXiv preprint arXiv:1905.12787*, 2019.

[41] GitHub. Github copilot, June 2022.

[42] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[43] Google. Preprocess data with tensorflow transform, November 2020.

[44] Google. Use a gpu in tensorflow, 2020.

[45] Google. grpc: A high-performance, open source universal rpc framework, 2021.

[46] Google. Tensorflow profiler, 2021.

[47] Google. tf.data: Build tensorflow input pipelines, 2021.

[48] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 689–706, 2022.

[49] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech Recognition with Deep Recurrent Neural Networks. In *In Proc. of International Conference on Acoustics, Speech and Signal Processing*, 2013.

[50] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[51] 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 (CVPR'16)*, pages 770–778, 2016.

[52] Yitao Hu, Swati Rallapalli, Bongjun Ko, and Ramesh Govindan. Olympian: Scheduling GPU Usage in a Deep Neural Network Model Serving System. In *Proc. of the 19th International Middleware Conference*, 2018.

[53] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020.

[54] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. volume 32, pages 103–112, 2019.

[55] huggingface. Bigscience model training launched, 2022.

[56] Zhouyuan Huo, Bin Gu, and Heng Huang. Training neural networks using features replay. *Advances in Neural Information Processing Systems*, 31, 2018.

[57] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic Space-Time Scheduling for GPU Inference. In *Proc. of Conference on Neural Information Processing Systems (NeurIPS)*, 2011.

[58] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. Multi-tenant GPU Clusters for Deep Learning Workloads: Analysis and implications. In *Tech. Rep.*, 2018.

[59] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in accelerating convolutional neural networks. In *Proceedings of the International Conference on Machine Learning (PMLR'18)*, 2018.

[60] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous gpu/cpu clusters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 463–479, 2020.

[61] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden,

Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proc. of the 44th Annual International Symposium on Computer Architecture(ISCA)*, 2017.

[62] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. In *Nature*, 2021.

[63] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *arXiv preprint arXiv:1909.11942*, 2019.

[64] Guillaume Leclerc, Andrew Ilyas, Logan Engstrom, Sung Min Park, Hadi Salman, and Aleksander Madry. FFCV: Accelerating training by removing data bottlenecks. https://github.com/libffcv/ffcv/, 2022. commit xxxxxxx.

[65] LeCun, Yann and Bengio, Yoshua and Hinton, Geoffrey. Deep learning. In *Nature*, 2015.

[66] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyung-Geun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish your training data: Reusing partially augmented samples for faster deep neural network training. In *USENIX Annual Technical Conference*, pages 537–550, 2021.

[67] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[68] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. 2020.

[69] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *Proceedings of the International Conference on Machine Learning (ICML'21)*, 2021.

[70] Yun Liang, Huynh Phung Huynh, Kyle Rupnow, Rick Siow Mong Goh, and Deming Chen. Efficient GPU Spatial-temporal Multitasking. In *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[71] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *arXiv preprint arXiv:1711.05101*, 2017.

[72] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training Deeper Models by GPU Memory Optimization on TensorFlow. In *Proc. of ML Systems Workshop in NIPS*, 2017.

[73] Meta. Democratizing access to large-scale language models with opt-175b, May 2022.

[74] Meta. Introducing pytorch fully sharded data parallel (fsdp) api. https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/, 2022.

[75] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. In *Proc. of Artificial Intelligence in the Age of Neural Networks and Brain Computing*, 2019.

[76] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. 2020.

[77] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, 2018.

[78] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, 2019.

[79] Mykola Novik. torch-optimizer – collection of optimization algorithms for PyTorch., 1 2020.

[80] NVIDIA. Maximizing unified memory performance in cuda, 2017.

[81] NVIDIA. Nvidia virtual gpu software documentation, 2018.

[82] NVIDIA. Gpudirect storage: A direct path between storage and gpu memory, 2019.

[83] NVIDIA. Cuda basic linear algebra subroutine library, January 2020.

[84] NVIDIA. Multi-process service, 2020.

[85] NVIDIA. The user manual for nvidia profiling tools for optimizing performance of cuda applications, 2020.

[86] NVIDIA. Nvlink and nvswitch - the building blocks of advanced multi-gpu communication—within and between servers., 2022.

[87] NVIDIA. CUDA Occupancy Calculator. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html, November 28, 2019.

[88] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-Serving: Flexible, High-performance ML Serving. In *In Proc. of Neural Information Processing Systems (NeurIPS), Long Beach, CA, USA.*, 2017.

[89] OpenAI. Scaling language model training to a trillion parameters using megatron, June 2020.

[90] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[91] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[92] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Dynamic resource management for efficient utilization of multitasking gpus. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[93] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic Differentiation in PyTorch. 2017.

[94] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[95] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *Proc. of Symposium on Operating Systems Principles (SOSP)*, 2019.

[96] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. *arXiv preprint arXiv:2212.04356*, 2022.

[97] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. In *OpenAI blog*, 2019.

[98] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[99] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20)*, 2020.

[100] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. 2021.

[101] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International Conference on Machine Learning*, pages 8821–8831. PMLR, 2021.

[102] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proc. of International Symposium on Microarchitecture (MICRO)*, 2016.

[103] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*, pages 1–13. IEEE, 2016.

[104] Sebastian Ruder. An Overview of Multi-task Learning in Deep Neural Networks. In *arXiv preprint arXiv:1706.05098*, 2017.

[105] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. Swarm parallelism: Training large models can be surprisingly communication-efficient. 2021.

[106] Max Ryabinin and Anton Gusev. Towards crowdsourced training of large neural networks using decentralized mixture-of-experts. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS'20)*, 2020.

[107] Max Ryabinin and Anton Gusev. Towards crowdsourced training of large neural networks using decentralized mixture-of-experts. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS'20)*, 2020.

[108] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. 2022.

[109] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[110] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Proceedings of the Advances in neural information processing systems (NeurIPS'18)*, 2018.

[111] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *arXiv preprint arXiv:1701.06538*, 2017.

[112] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[113] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

[114] Ilia Shumailov, Zakhar Shumaylov, Dmitry Kazhdan, Yiren Zhao, Nicolas Papernot, Murat A Erdogdu, and Ross J Anderson. Manipulating sgd with data ordering attacks. *Advances in Neural Information Processing Systems*, 34:18021–18032, 2021.

[115] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling Preemptive Multiprogramming on GPUs. In *Proc. of International Symposium on Computer Architecture (ISCA)*, 2014.

[116] DeepSpeed Team. Deepspeed: Accelerating large-scale model inference and training via system optimizations and compression, June 2021.

[117] DeepSpeed Team. Zero-offload, June 2021.

[118] Learning@home team. Hivemind: a Library for Decentralized Deep Learning. https://github.com/learning-at-home/hivemind, 2020.

[119] Stanford University. National research cloud, 2022.

[120] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. volume 30, 2017.

[121] Mohamed Wahib and Naoya Maruyama. Scalable Kernel Fusion for Memory-bound GPU Applications. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[122] Chengyi Wang, Sanyuan Chen, Yu Wu, Ziqiang Zhang, Long Zhou, Shujie Liu, Zhuo Chen, Yanqing Liu, Huaming Wang, Jinyu Li, et al. Neural codec language models are zero-shot text to speech synthesizers. *arXiv preprint arXiv:2301.02111*, 2023.

[123] Hao Wang, Di Niu, and Baochun Li. Dynamic and decentralized global analytics via machine learning. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'28)*, 2018.

[124] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi. Exploiting Concurrent Kernel Execution on Graphic Processing Units. In *Proc. of High performance computing and simulation (HPCS)*, 2011.

[125] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous Multikernel GPU: Multi-tasking throughput Processors via Fine-grained Sharing. In *Proc. of IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[126] Wikimedia. A complete copy of all wikimedia wikis, in the form of wikitext source and metadata embedded in xml., September 2022.

[127] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and Exploiting Flexible Task Assignment on GPU through SM-centric Program Transformations. In *Proc. of International Conference on Supercomputing (SC)*, 2015.

[128] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. Flep: Enabling flexible and efficient preemption on gpus. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[129] Xiaofeng Wu, Jia Rao, Wei Chen, Hang Huang, Chris Ding, and Heng Huang. Switchflow: preemptive multitasking for deep learning. In *Proceedings of the 22nd International Middleware Conference (Middleware'21)*, 2021.

[130] 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 *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[131] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on gpu clusters for deep learning. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[132] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. Gspmd: general and scalable parallelization for ml computation graphs. In *arXiv preprint arXiv:2105.04663*, 2021.

[133] Chao Yu, Yuebin Bai, Hailong Yang, Kun Cheng, Yuhao Gu, Zhongzhi Luan, and Depei Qian. SMGuard: A Flexible and Fine-grained Resource Management Framework for GPUs. In *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[134] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, Haoran Zhang, and Jie Zhao. Oneflow: Redesign the distributed deep learning framework from scratch, 2021.

[135] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. In *arXiv preprint arXiv:2205.01068*, 2022.

[136] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 1042–1057, 2022.

[137] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *arXiv preprint arXiv:2201.12023*, 2022.