Gandiva: Introspective Cluster Scheduling for Deep Learning

Wencong Xiao, *Beihang University & Microsoft Research;* **Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, and Nipun Kwatra,** *Microsoft Research;* **Zhenhua Han,** *The University of Hong Kong & Microsoft Research;* **Pratyush Patel,** *Microsoft Research;* **Xuan Peng,** *Huazhong University of Science and Technology & Microsoft Research;* **Hanyu Zhao,** *Peking University & Microsoft Research;* **Quanlu Zhang, Fan Yang, and Lidong Zhou,** *Microsoft Research*

https://www.usenix.org/conference/osdi18/presentation/xiao

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-931971-47-8

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Gandiva: Introspective Cluster Scheduling for Deep Learning

Wencong Xiao[†⋆*], Romil Bhardwaj[⋆*], Ramachandran Ramjee[⋆], Muthian Sivathanu[⋆], Nipun Kwatra[⋆], Zhenhua Han[◇⋆], Pratyush Patel[⋆], Xuan Peng[‡⋆], Hanyu Zhao[§⋆], Quanlu Zhang[⋆], Fan Yang[⋆], Lidong Zhou[⋆]
[†]*Beihang University,* [⋆]*Microsoft Research,* [◇]*The University of Hong Kong,*
[‡]*Huazhong University of Science and Technology,* [§]*Peking University*

Abstract

We introduce Gandiva, a new cluster scheduling framework that utilizes domain-specific knowledge to improve latency and efficiency of training deep learning models in a GPU cluster.

One key characteristic of deep learning is feedback-driven exploration*, where a user often runs a set of jobs (or a multi-job) to achieve the best result for a specific mission and uses early feedback on accuracy to dynamically prioritize or kill a subset of jobs; simultaneous early feedback on the entire multi-job is critical. A second characteristic is the heterogeneity of deep learning jobs in terms of resource usage, making it hard to achieve best-fit a priori. Gandiva addresses these two challenges by exploiting a third key characteristic of deep learning:* intra-job predictability*, as they perform numerous repetitive iterations called mini-batch iterations. Gandiva exploits intra-job predictability to time-slice GPUs efficiently across multiple jobs, thereby delivering low-latency. This predictability is also used for introspecting job performance and dynamically migrating jobs to better-fit GPUs, thereby improving cluster efficiency.*

We show via a prototype implementation and micro-benchmarks that Gandiva can speed up hyper-parameter searches during deep learning by up to an order of magnitude, and achieves better utilization by transparently migrating and time-slicing jobs to achieve better job-to-resource fit. We also show that, in a real workload of jobs running in a 180-GPU cluster, Gandiva improves aggregate cluster utilization by 26%, pointing to a new way of managing large GPU clusters for deep learning.

1 Introduction

All ~~men~~ schedulers make mistakes; only the wise learn from their mistakes.

-Winston Churchill

An increasingly popular computing trend over the last few years is deep learning [32]; it has already had significant impact; *e.g.*, on widely-used personal products for voice and image recognition, and has significant potential to impact businesses. Hence, it is likely to be a vital and growing workload, especially in cloud data centers.

However, deep learning is compute-intensive and hence heavily reliant on powerful but expensive GPUs; a GPU VM in the cloud costs nearly 10x that of a regular VM. Cloud operators and large companies that manage clusters of tens of thousands of GPUs rely on cluster schedulers to ensure efficient utilization of the GPUs.

Despite the importance of efficient scheduling of deep learning training (DLT) jobs, the common practice today [12, 28] is to use a traditional cluster scheduler, such as Kubernetes [14] or YARN [50], designed for handling big-data jobs such as MapReduce [17]; a DLT job is treated simply as yet another big-data job that is allocated a set of GPUs at job startup and holds exclusive access to its GPUs until completion.

In this paper, we present *Gandiva*, a new scheduling framework that demonstrates that a significant increase in cluster efficiency can be achieved by *tailoring* the scheduling framework to the unique characteristics of the deep learning workload.

One key characteristic of DLT jobs is *feedback-driven exploration* (Section 2). Because of the inherent trial-and-error methodology of deep learning experimentation, users typically try several configurations of a job (a *multi-job*), and use early feedback from these jobs to decide whether to prioritize or kill some subset of them. Such conditional exploration, called hyper-parameter search, can either be manual or automated [10, 33, 41]. Traditional schedulers run a subset of jobs to completion while queueing others; this model is a misfit for multi-jobs, which require simultaneous early feedback on all jobs within the multi-job. Also, along with multi-jobs, other DLT jobs that have identified the right hyper-parameters, run for several hours to days, leading to

[*]The first two authors have equal contribution. This work is done while Wencong Xiao, Zhenhua Han, Xuan Peng, and Hanyu Zhao are interns in Microsoft Research.

head-of-line-blocking, as long-running jobs hold exclusive access to the GPUs until completion, while multi-jobs depending on early feedback wait in queue. Long queueing times force users to either use reserved GPUs, or demand cluster over-provisioning, thus reducing cluster efficiency.

Second, like any other cluster workload, DLT jobs are heterogeneous because of the diverse application domains they target. Jobs widely differ in terms of memory usage, GPU core utilization, sensitivity to interconnect bandwidth, and/or interference from other jobs. For example, certain multi-GPU DLT jobs may perform much better with affinitized GPUs, while other jobs may not be as sensitive to affinity (Section 3). A traditional scheduler that treats a job as a black-box will hence achieve sub-optimal cluster efficiency.

To address the twin problems of high latency and low efficiency, *Gandiva* exploits a powerful property of DLT jobs: *intra-job predictability* (Section 3). A job is comprised of millions of similar, clearly separated mini-batch iterations. For example, the GPU RAM usage of a DLT job follows a cyclic pattern aligned with mini-batch boundaries, usually with more than 10x difference in GPU RAM usage within a mini-batch. *Gandiva* exploits this cyclic predictability to implement efficient *application aware time-slicing*; in effect, it re-defines the atom of scheduling from a *job* to automatically-partitioned *micro-tasks*. This enables the cluster to over-subscribe DLT jobs and provide *early feedback* through time-slicing to all DLT jobs, including all jobs that are part of a multi-job.

Gandiva also uses the predictability to perform *profile-driven introspection*. It uses the mini-batch progress rate to introspect its decisions continuously to improve cluster efficiency (Section 4). For example, it packs multiple jobs on the same GPU only when they have low memory and GPU utilization; it dynamically migrates a communication intensive job to more affinitized GPUs; it also opportunistically "grows" the degree of parallelism of a job to make use of spare resources, and shrinks the job when the spare resources go away. The introspection policy we presently implement is a stateful trial-and-error policy that is feasible because of the predictability and the limited state space of options we consider.

Beyond the specific introspection and scheduling policy evaluated in this paper, the *Gandiva* framework provides the following APIs that any DLT scheduling policy can leverage: (a) efficient suspend-resume or time-slicing, (b) low-latency migration, (c) fine-grained profiling, (d) dynamic intra-job elasticity, and (e) dynamic prioritization. The key to making these primitives efficient and practical is the co-design approach of *Gandiva* that spans across both the scheduler layer and the DLT toolkit layer such as Tensorflow [8] or PyTorch [38]. Traditional schedulers, for a good reason, treat a job as a black-box. However, by exploiting the dedicated nature of GPU clusters, *Gandiva* customizes the scheduler to the specific workload of deep learning, thus providing the scheduler more visibility and control into a job, while still achieving generality to arbitrary DLT jobs.

We have implemented *Gandiva* by modifying two popular frameworks, PyTorch and Tensorflow, to provide the necessary new primitives to the scheduler, and also implemented an initial scheduling policy manager on top of Kubernetes and Docker containers (Section 5). We evaluate *Gandiva* on a cluster of 180 heterogeneous GPUs and show, through micro-benchmarks and real workloads, that (i) *Gandiva* improves the efficiency of cluster scheduling by up to 26%, and (ii) *Gandiva* is reactive enough to time-slice multiple jobs dynamically on the same GPU, reducing the time to early feedback by as much as 77%. We also show that, for a popular hyper-parameter search technique [10], *Gandiva* improves the overall completion time of the hyper-parameter search by up to an order of magnitude while using same resources (Section 6).

The key contributions of the paper are as follows.

- We illustrate various unique characteristics of the deep learning workflow and map it to specific requirements needed for cluster scheduling.
- We identify generic primitives that can be used by a DLT job scheduling policy, and provide application-aware techniques to make primitives such as time-slicing and migration an order of magnitude more efficient and thus practical by leveraging DL-specific knowledge of intra-job periodicity.
- We propose and evaluate a new introspective scheduling framework that utilizes domain-specific knowledge of DLT jobs to refine its scheduling decision continuously, thereby significantly improving early feedback time and delivering high cluster efficiency.

2 Background

Deep learning is a type of representation learning that automatically infers features from raw data in order to accomplish tasks such as image classification or language translation [32]. Deep learning may be supervised (data with labels) or unsupervised (data only). In either case, the representation is a *deep neural network model* with parameters called weights. These weights are carefully arranged in layers and number typically in the millions. These model weights are learned through *training*.

Deep learning training operates on a few samples of data at a time called a *mini-batch*. It computes a set of scores for each mini-batch by performing numerical

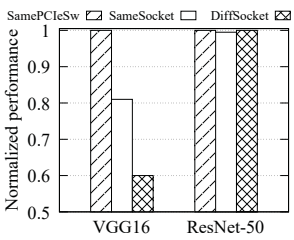


Figure 1: Intra-server locality.

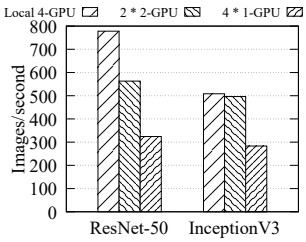


Figure 2: Inter-server locality.

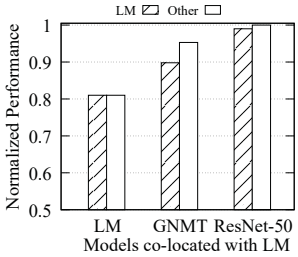


Figure 3: 1-GPU interference.

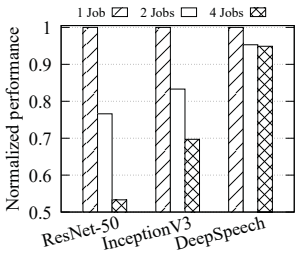


Figure 4: NIC interference.

computations using the model weights, called the *forward pass*. Based on the desired task, an objective function is defined that measures an error between the computed scores and desired scores. The error is populated via a *backward pass* over the model, where it first computes a gradient for each weight (i.e., the impact of each weight on the error) and then applies a negative of the gradient, scaled by a parameter called the learning rate, to each weight to decrease the error. Both the forward and backward passes typically involve billions of floating point operations and thus leverage GPUs. Each forward-backward pass is called a mini-batch iteration. Typically, millions of such iterations are performed on large datasets to achieve high task accuracy.

Feedback-driven exploration. One pre-requisite for achieving high accuracy is model selection. Discovery of new models such as ResNet [24] or Inception [46] is mostly a trial-and-error process today, though ways to automate it is an active area of research [36].

Apart from the model structure, there are a number of parameters, called *hyper-parameters*, that also need to be specified as part of the DLT job. Hyper-parameters include the number of layers/weights in the model, mini-batch size, learning rate, etc. These are typically chosen today by the user based on domain knowledge and trial-and-error, and can sometimes even result in early training failure. Thus, *early-feedback* on DLT jobs is critical, especially in the initial stages of training.

Multi-job. Once the user has identified a particular model to explore further, the user typically performs *hyper-parameter search* to improve task accuracy. This can be done using various searching techniques over the space of the hyper-parameters; that is, the user generates *multiple DLT jobs* or multi-jobs, each performing full training using one set of hyper-parameters or configuration. Because users typically explore hundreds of such configurations, this process is computationally expensive. Thus, sophisticated versions of hyper-parameter searches are available in the literature, such as HyperOpt [10] and Hyperband [33]. For example, Hyperband might initially spawn 128 DLT jobs and, in each round (*e.g.*, 100 mini-batch iterations), kill half of the jobs with the lowest accuracy. Again, for these algorithms, *early feedback* on the entire set of jobs is crucial because they would be unable to make effective training decisions otherwise.

3 DLT Job Characteristics

In this section, we motivate the design of *Gandiva* by highlighting several unique characteristics of DLT jobs.

3.1 Sensitivity to locality

The performance of a multi-GPU DLT job depends on the affinity of the allocated GPUs. Different DLT jobs exhibit different levels of sensitivity to inter-GPU affinity. Even for GPUs on the same machine, we observe different levels of inter-GPU affinity due to asymmetric architecture: two GPUs might be located in different CPU sockets (denoted as DiffSocket), in the same CPU socket, but on different PCIe switches (denoted as SameSocket), or on the same PCIe switch (denoted as SamePCIeSw).

Figure 1 shows different sensitivity to *intra-server locality* for two models VGG16 [44] and ResNet-50 [24]. When trained with two P100 GPUs using Tensorflow, VGG16 suffers greatly under bad locality. With the worst locality, when two GPUs are located in different CPU sockets, VGG16 achieves only 60% of the best locality config, where two GPUs are placed under the same PCIe switch. On the other hand, the ResNet-50 is not affected by GPU locality in this setting. This is because VGG16 is a larger neural model than ResNet-50, hence the model synchronization in each mini-batch incurs a higher communication load on the underlying PCIe bus.

We observe similar trends in a distributed setting. Figure 2 shows the performance of a 4-GPU Tensorflow job running with different *inter-server locality*, training ResNet-50 and InceptionV3 [46] models. Even when interconnected with a 40G InfiniBand network, the performance difference is clearly seen when the job is assigned to 4 GPUs, where they are evenly scattered across 4 servers (denoted as 4*1-GPU), 2 servers (denoted as 2*2-GPU), and all in one server (denoted as local 4-GPU), though the sensitivity to locality of the two models is different.

Thus, a DLT scheduler has to take into account a job's sensitivity to locality when allocating GPUs.

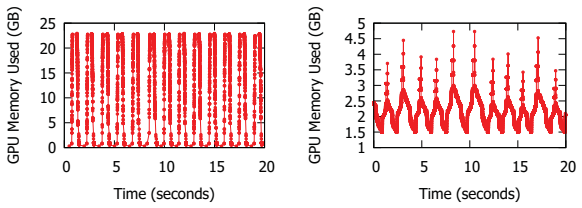
3.2 Sensitivity to interference

When running in a shared execution environment, DLT jobs might interfere with each other due to resource contention. We again observe that different DLT jobs exhibit different degrees of interference.

Interference exists even for single-GPU jobs. When placing a Language Model [56] job (marked as LM) with another job under the same PCI-e switch, Figure 3 shows the performance degradation due to *intra-server* interference. When two LMs run together, both jobs suffer 19% slowdown. However, ResNet-50 does not suffer from GPU co-location with LM. Neural Machine Translation (GNMT) [51] exhibits a modest degree of interference with LM. Similarly, we also observe various degrees of interference for multi-GPU training with different types of training models. We omit the result due to space limitation.

Figure 4 shows *inter-server interference* on two 4-GPU servers that are connected with a 40G InfiniBand network. When running multiple 2-GPU jobs, where each GPU is placed on different server, ResNet-50 shows up to 47% slowdown, InceptionV3 shows 30% slowdown, while DeepSpeech [23] only shows 5% slowdown.

In summary, popular deep learning models across different application domains such as vision, language, and speech demonstrate different levels of sensitivity to locality and interference. To cater to these challenges, *Gandiva* leverages a key characteristic of DLT jobs, which we elaborate next.



(a) ResNet50/Imagenet (b) GNMT/WMT'14 En-De

Figure 5: GPU memory usage during training.

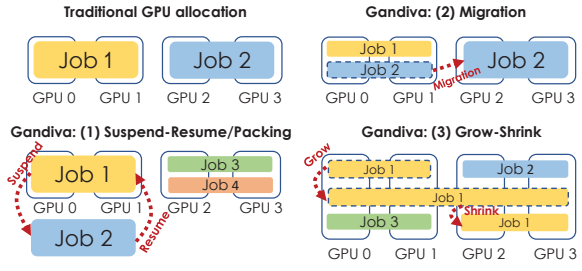


Figure 6: GPU usage options in Gandiva.

3.3 Intra-job predictability

A DLT job consists of numerous mini-batch iterations. The total GPU memory used[1] during a 20s snapshot of training on ImageNet data when using ResNet-50 model [24] on four K80 GPUs is shown in Figure 5(a). The GPU memory used clearly follows a cyclic pattern. Each of these cycles corresponds to the processing of a single mini-batch (about 1.5s), with the memory increasing during the forward pass and decreasing during the backward pass. The maximum and minimum GPU memory used is 23GB and 0.3GB, respectively, or a factor of 77x. This ratio scales with the mini-batch size (typically between 16 to 256; 128 in this case).

The total GPU memory used during a 20s snapshot of training on WMT'14 English German language dataset when using GNMT model [51] on one K80 GPU is shown in Figure 5(b). While the mini-batch iterations are not identical to each other as in the ImageNet example (due to differing sentence lengths and the use of dynamic graphs in PyTorch), the graph has a similar cyclic nature. The difference between maximum and minimum is smaller (3x) primarily due to larger model (0.4GB) and smaller mini-batch size (16 in this example).

Apart from image and language models shown here, other training domains such as speech, generative adverserial networks (GANs), and variational auto-encoders all follow a similar cyclic pattern (not shown due to space limitation) since the core of training is the gradient descent algorithm performing many mini-batch iterations.

Leveraging predictability. This characteristic behavior is exploited in *Gandiva* in multiple ways. First, a DLT job can be automatically split into mini-batch iterations and a collection of these iterations over 60 seconds, say a micro-task, forms a scheduling interval. Second, by performing the suspend operation at the minimum of the memory cycle, the amount of memory to be copied from GPU to be saved in CPU can be significantly reduced, thereby enabling suspend/resume and migration to be an order of magnitude more efficient than a naïve implementation. Third, the mini-batch progress rate can be profiled and used as a proxy to evaluate the effectiveness of applying mechanisms such as packing or migration.

4 Design

High latency and low utilization in today's cluster arises because DLT jobs are assigned a *fixed set* of GPUs *exclusively* (Figure 6). Exclusive access to GPUs causes

[1]This is actual GPU memory used. Toolkits like PyTorch/Tensorflow use caching to avoid expensive GPU memory (de)allocations.

head-of-line blocking, preventing early feedback and resulting in high queuing times for incoming jobs. Exclusive access to a fixed set of GPUs also results in low GPU utilization when jobs are unable to utilize their assigned GPUs fully.

4.1 Mechanisms

In *Gandiva*, we address these inefficiencies by removing the exclusivity and fixed assignment of GPUs to DLT jobs in three ways (Figure 6). First, during overload, instead of waiting for current jobs to depart, *Gandiva* allows incoming jobs to time-share GPUs with existing jobs. This is enabled using a custom suspend-resume mechanism tailored for DLT jobs along with selective packing. Second, *Gandiva* supports efficient migration of DLT jobs from one set of GPUs to another. Migration allows time-sliced jobs to migrate to other (recently vacated) GPUs or for de-fragmentation of the cluster so that incoming jobs are assigned GPUs with good locality. Third, *Gandiva* supports a GPU grow-shrink mechanism so that idle GPUs can be used opportunistically. In order to support these mechanisms efficiently and enable effective resource management, *Gandiva* introspects DLT jobs by continuously profiling their resource usage and estimating their performance. We now describe each of these mechanisms.

Suspend-Resume and Packing. Suspend-resume is one mechanism *Gandiva* uses to *remove exclusivity* of a set of GPUs to a DLT job. Modern operating systems support efficient suspend-resume for CPU process time-slicing. *Gandiva* leverages this mechanism and adds custom support for GPU time-slicing.

As shown in Figure 5(a), usage of GPU memory by DLT jobs has a cyclic pattern with as much as 77x difference between the minimum and maximum memory usage. The key idea in *Gandiva* is to exploit this cyclic behavior and suspend-resume DLT jobs when their GPU memory usage is at their lowest. Thus, when a suspend call is issued, the DLT toolkit waits until the minimum of the memory usage cycle, copies the objects stored in the GPU to the CPU, releases all its GPU memory allocations (including cache), and then invokes the classic CPU suspend mechanism. Later, when the CPU resumes the job, the DLT framework first allocates appropriate GPU memory, copies the stored objects back to the GPU, and then resumes the job.

Suspend-resume may also initiate a change of GPU within the same server (*e.g.*, in the case of six 1-GPU jobs time-sharing 4-GPUs). While changing GPU is expensive, we hide this latency from the critical path. As we show in our evaluation (Section 6.1), for typical image classification jobs, suspend-resume together can be accomplished in under 100ms, while for large language translation jobs suspend-resume can take up to 1s. Given a time-slicing interval of 1 minute, this amounts to an overhead of 2% or less.

Note that suspend in *Gandiva* may be delayed by at most a mini-batch interval of the DLT job (typically, a few seconds or less), but we believe this is a worthwhile trade-off as it results in significantly less overhead due to the reduced GPU-CPU copy cost and less memory used in the CPU. Further, useful work is accomplished during this delay. The scheduler keeps track of this delay and adjusts the time-slicing interval accordingly for fairness.

An alternative to suspend-resume for time-slicing is to run multiple DLT jobs on a GPU *simultaneously* and let the GPU time-share the jobs. We call this *packing*. Packing in GPU is efficient only when the packed jobs do not exceed the GPU resources (cores, memory) and do not adversely impact each other. If jobs interfere, packing can be significantly worse than suspend-resume (Section 6.1). We use profiling to monitor the resource and progress of DLT jobs when they have exclusive access. If two jobs are identified as candidates for packing, we pack them together and continue monitoring them. If a given packing results in adverse impact on jobs' performance, we unpack those jobs and revert to suspend-resume.

Migration. Migration is the mechanism *Gandiva* uses to *change the set* of GPUs assigned to a DLT job. Migration is useful in several situations such as i) moving time-sliced jobs to vacated GPUs anywhere in the cluster; ii) migrating interfering jobs away from each other; iii) de-fragmentation of the cluster so that incoming jobs get GPUs with good locality.

We evaluate two approaches for tackling DLT process state migration. In the first approach, we leverage a generic process migration mechanism such as CRIU [1]. Because CRIU by itself does not support migration of processes that use the GPU device, we first checkpoint GPU objects and remove all GPU state from the process before CRIU is invoked. Because CRIU checkpoints and restores the entire process memory, the size of the checkpoint is on the order of GBs for these DLT jobs using PyTorch. Thus, the resulting migration overhead is about 8-10s for single GPU jobs and higher for multi-GPU jobs.

The second approach we consider is the use of DLT jobs that are checkpoint-aware. DLT frameworks such as Tensorflow already support APIs (*e.g.*, `tensorflow.train.saver`) that allow automatic checkpoint and restore of models. This API is used today to ensure that long running jobs do not have to be rerun due to server failures. We extend the framework to support migration of such jobs. By *warming up* the destination before migration and only migrating the necessary training state, we can reduce the migration over-

head to as little as a second or two (Section 6.1). With either approach, we find that the overhead of inter-server migration is worthwhile compared to the benefits it provides in terms of higher overall GPU utilization.

Grow-Shrink. The third mechanism that *Gandiva* uses to remove the exclusivity of GPUs to a DLT job is grow-shrink. This mechanism primarily targets situations when the cluster may not be fully utilized, say, late at night. The basic idea is to grow the number of GPUs available to a job opportunistically during idle times and correspondingly also shrink the number of GPUs available when the load increases.

Many DLT jobs, especially in the image domain, see linear performance scaling as the number of GPUs is increased. *Gandiva* applies this mechanism only to those DLT jobs that specifically declare that they are adaptive enough to take advantage of these growth opportunities. When multiple DLT jobs fit this criteria, *Gandiva* uses profiling information, discussed next, to estimate each job's progress rate and then allocate GPUs accordingly.

Profiling. Like any scheduler, *Gandiva* monitors resource usage such as CPU and GPU utilization, CPU/GPU memory, etc. However, what is unique to *Gandiva* is that it also *introspects* DLT jobs in an application-aware manner to estimate their rate of progress. This introspection exploits the regular pattern exhibited by DLT jobs (Section 3) and uses the periodicity to estimate their progress rate.

Gandiva estimates a DLT job's `mini_batch_time`, the time to do one forward/backward pass over a batch of input data, as the time taken between two minimums of the GPU memory usage cycles (Figure 5(a)). Because DLT jobs typically perform millions of such mini batch operations in their lifetime, the scheduler compares the `mini_batch_time` of a DLT prior to and post a scheduling decision to determine its effectiveness.

For example, consider the example of packing two DLT jobs in a GPU described earlier. By comparing the `mini_batch_time` of each of the two DLT jobs before and after packing, *Gandiva* can decide whether packing is effective. Without such profiling, in order to make a packing decision, one would have to model not only the two DLT jobs' performance on various GPUs but also the various ways in which they may interfere with each other (*e.g.*, caches, memory bandwidth, etc.), a non-trivial task as evidenced by the varied performance of packing we see in Section 6.1.

4.2 Scheduling Policy

Definitions: Before we describe the details of the scheduler, we define some terminology. DLT jobs are encapsulated in containers (Section 5) and include the number of GPUs required, their priority (can be dynamically changed), and a flag indicating if the job is capable of grow-shrink. We assume the number of GPUs requested by a job is a power of two (typical for DLT jobs today). A cluster is composed of one or more servers, with each server having one or more GPUs. Further, we assume a dedicated GPU cluster for DLT jobs [28, 12].

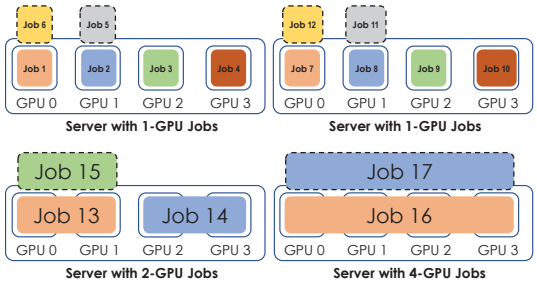


Figure 7: Scheduling example in a 16-GPU Cluster.

We define the *height* of a server as $\lceil M/N \rceil$, where M is the number of allocated GPUs and N is the number of total GPUs. Thus, the suspend/resume mechanism will only be used when the height of a server exceeds one. The height of a cluster is defined as the maximum height of all its servers. *Overload* occurs when the height of the cluster is greater than one; i.e., the sum of requested/allocated GPUs of all jobs is greater than the total number of GPUs. We define the *affinity* of a server as the type of jobs (based on GPUs required) assigned to that server. For example, initially servers have affinity of zero and, if a job that requires two GPUs is assigned to a server, the affinity of that server is changed to two. This parameter is used by the scheduler to assign jobs with similar GPU requirements to the same server.

Goals: The primary design goal of the *Gandiva* scheduler is to provide ***early feedback*** to jobs. In prevalent schedulers, jobs wait in a queue during overload. In contrast, *Gandiva* supports over-subscription by allocating GPUs to a new job immediately and using the suspend-resume mechanism to provide early results. A second design goal is ***cluster efficiency***. This is achieved through a continuous optimization process that uses profiling and a greedy heuristic that takes advantage of mechanisms such as packing, migration, and grow-shrink. ***Cluster-level fairness*** is not a design goal in *Gandiva*. While we believe achieving long-term fairness at the cluster level is feasible using the *Gandiva* mechanisms, in this paper, we focus only on providing fairness among jobs at each server using the suspend-resume mechanism and leave cluster-level fairness to future work.

To achieve these goals, the *Gandiva* scheduler operates in two modes: *reactive and introspective*. By reactive mode, we refer to when the scheduler reacts to events such as job arrivals, departures, machine failures etc. By introspective mode, we refer to a continuous process where the scheduler aims to improve cluster utiliza-

Algorithm 1 getNodes(in job, out nodes)

```
1:  nodes0 ← findNodes(job.gpu, affinity ← job.gpu)
2:  nodes1 ← minLoadNodes(node0)
3:  nodes2 ← findNodes(jog.gpu, affinity ← 0)
4:  nodes3 ← findNodes(job.gpu)
5:  if nodes1 and height(nodes1) < 1:
6:      return nodes1            // Same affinity with free GPUs
7:  if nodes2 and numGPUs(nodes2) ≥ job.gpu:
8:      return nodes2                // Unallocated GPU servers
9:  if nodes3:
10:     return nodes3                // Relax affinity constraint
11: elif nodes1:
12:     return nodes1                // Allow over-subscription
13: else:
14:     enqueue(job)                                  // Job queued
```

tion and job completion time. Note that the scheduler can be operating in both modes at the same time. We discuss each of these modes next.

4.2.1 Reactive Mode

The reactive mode is designed to take care of events such as job arrivals, departures, and machine failures. Conventional schedulers operate in this mode. Here we discuss only our job placement policy since we follow the conventional approach for failure handling.

When a new job arrives, the scheduler allocates servers/GPUs for the job. The node allocation policy used in *Gandiva* is shown in Algorithm 1. *findNodes* is a function to return the node candidates that satisfy the job request with an optional parameter for affinity constraint. Initially, *Gandiva* tries to find nodes with the same affinity as the new job and, among those, ones with the minimum loads. If such nodes exist and their height is less than one (lines 5–6), that node is assigned. Otherwise, *Gandiva* tries to find and assign un-affinitized nodes (lines 7–8). If no such free servers are available, the third option is to look for nodes with free GPUs while ignoring affinity (lines 9–10). This may result in fragmented allocation across multiple nodes but, as we shall see later, migration can be used for defragmentation. If none of the above work, it implies that no free GPUs are available in the cluster. In this case, if nodes with the same affinity exist, they are used with suspend-resume (lines 11–12); if not, the job is queued (lines 13–14).

For example, as shown in Figure 7, jobs that require 1-GPU are placed together but jobs that require 2 or 4 GPUs are placed on different servers. Further, we try to balance the over-subscription load on each of the servers by choosing the server with the minimum load (*e.g.*, six 1-GPU jobs on each of the two servers in the figure).

Conventional schedulers will use job departures to pick the next job from the waiting queue for placement.

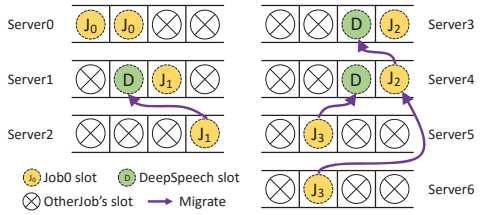


Figure 8: Job migration in a shared cluster.

In addition, in *Gandiva*, we check whether the height of the cluster can be reduced; e.g., by migrating a job that is suspended to the newly vacated GPU. This job could be from the same server or from any other server in the cluster. Finally, job departures can also trigger migrations for improving locality, as discussed in the next section.

Gandiva's job placement policy takes into account two factors. First, unlike conventional schedulers, *Gandiva* allows over-subscription. When a server is over-subscribed, we do weighted round-robin scheduling to give each job its fair time-share. Second, unlike today's schedulers, where GPU allocation is a one-time event at job arrival, *Gandiva* uses the introspective mode, discussed next, to improve cluster utilization continuously. Thus, *Gandiva* relies on a simple job placement policy to allocate GPU resources quickly to new jobs, thereby enabling early feedback.

4.2.2 Introspective Mode

In the introspective mode, *Gandiva* continuously monitors and optimizes placement of jobs to GPUs in the cluster to improve the overall utilization and the completion time of DLT jobs.

Packing. Packing is considered only during overload. The basic idea behind packing is to run two or more jobs simultaneously on a GPU to increase efficiency. If the memory requirements of the packing jobs combined are higher than GPU memory, the overhead of "paging" from CPU memory is significantly high [16] that packing is not effective. When the memory requirements of two or more jobs are smaller than GPU memory, packing still may not be more efficient than suspend-resume as we show in Section 6.1. For example, for some DLT jobs, packing increases efficiency, while for others packing can be worse than suspend-resume.

Analytically modeling performance of packing is a challenging problem given the heterogeneity of DLT jobs. Instead, *Gandiva* relies on a greedy heuristic to pack jobs. When jobs arrive, we always run them in exclusive mode using suspend-resume and collect profiling information (GPU utilization, memory and job progress rate). Based on the profiling data, the scheduler maintains a list of jobs sorted by their GPU utilization. The scheduler greedily picks the job with the lowest GPU uti-

lization and attempts to pack it on a GPU with the lowest GPU utilization. We only do this when the combined memory utilization of the packed jobs do not exceed the overall memory of the GPU. Packing is deemed successful when the total throughput of packed jobs is greater than time-slicing. If packing is unsuccessful, we undo the packing and try the next lowest utilization GPU. If the packing is successful, we find the next lower utilization job and repeat this process. Based on our evaluation, we find that this simple greedy heuristic achieves 26% efficiency gains.

Migration. GPU locality can play a significant role in the performance of some jobs (Section 3.1). In *Gandiva*, we use migration to improve locality whenever a job departs and also as a background process to "defrag" the cluster. To improve locality, we pick jobs that are not co-located and try to find a new co-located placement. Figure 8 illustrates an example from a cluster experiment (Section 6.4). When a multi-job with 4 jobs that requires 2-GPUs each was scheduled, it had poor GPU affinity; only J_0's two GPUs are colocated with the other 3 jobs in the multi-job (J_1, J_2, and J_3,) assigned to separated GPUs. Three minutes later, a background training job, DeepSpeech, completes and releases its 8 GPUs. Three of the 8 GPUs, marked as D in Figure 8 in three different servers (server 1, 3, and 4), can improve the training efficiency of the multi-job. *Gandiva* hence initiates the migration process, relocating J_1, J_2, and J_3 to colocated GPUs. For de-fragmentation, we pick the server with the most free GPUs among all non-idle ones. We then try to move the jobs running on that server to others. The job will be migrated to another server with fewer free GPUs, as long as there is negligible performance loss. We repeat this until the number of free GPUs on every non-idle server is less than a threshold (3 out of 4 in our experiments) or if no job will benefit from migration.

Grow-shrink. Grow-shrink is only triggered when the cluster is under-utilized and the DLT jobs specifically identify themselves as amenable to grow-shrink. In our current system, we only grow jobs to use up to the maximum number of GPUs available in a single server. Further, we trigger growth only after an idle period to avoid thrashing and shrink immediately when a new job might require the GPUs.

Time-slicing. Finally, we support round robin scheduling in each server to time-share GPUs fairly (Section 6.1). When jobs have multiple priority levels, higher priority jobs will never be suspended to accommodate lower priority jobs. If a server is fully utilized with higher priority jobs, the lower priority job will be migrated to another server, if feasible.

5 Implementation

DLT jobs are encapsulated as Docker containers containing our customized versions of DL toolkits and a *Gandiva* client. These jobs are submitted to a Kubernetes [14] system. *Gandiva* also implements a custom scheduler that then schedules these jobs.

5.1 Scheduler

Gandiva consists of a custom central scheduler and also a client component that is part of every DLT job container. The scheduler is just another container managed by Kubernetes. Kubernetes is responsible for overall cluster management, while the *Gandiva* scheduler manages the scheduling of DLT jobs. The *Gandiva* scheduler uses the Kubernetes API to get cluster node and container information and, whenever a new container is submitted, the scheduler assigns it to one or more of the GPUs in the cluster based on the scheduling policy.

When a container is scheduled on a node, initially only the *Gandiva* client starts executing. It then polls the *Gandiva* scheduler to identify which GPUs to make available for the DLT job and also controls the execution of the DLT job using suspend/resume and migrate commands. While scheduling of all the GPUs in our cluster is fully controlled by the central scheduler, a hierarchical approach may be needed if scalability becomes a concern.

5.2 Modifications to DL toolkits

In the interest of space, we describe only the time-slicing implementation for PyTorch and the migration implementation for Tensorflow.

PyTorch time-slicing. The *Gandiva*client issues a SIGTSTP signal to indicate that the toolkit must suspend the process. It also indicates whether or not the resume should occur in a new GPU via an in-memory file. Upon receiving the signal, the toolkit sets a suspend flag and executes the suspend only at the end of a mini-batch boundary.

In Tensorflow, a define-and-run toolkit, the mini-batch boundaries are easily identified (end of `session.run()`). In PyTorch, a define-by-run toolkit, we identify the mini-batch boundary by tracking GPU memory usage cycles as part of PyTorch's GPU memory manager (THCCachingAllocator) and looking for a cycle minimum whenever GPU memory is freed.

Once the minimum is detected, the toolkit i) copies all stored objects from GPU to CPU, ii) frees up GPU allocations, and iii) suspends the process. When *Gandiva* client issues a SIGCONT signal, the toolkit allocates GPU memory, copies stored objects from CPU to GPU, and resumes the process. To handle device address

change on resume, we track GPU objects in the toolkit and *patch* them with the new addresses. Changing GPU involves calling cudaDeviceReset and CudaInit, which can take 5-10s. We hide this latency by performing these actions in the background while "suspended".

Tensorflow migration. We make changes to Tensorflow (TF) with 400+ lines of Python/C++ code. With 200+ line of additional code, we deploy a *Migration Helper* on each server to support on-demand checkpointing and migration. When receiving a migration command from the scheduler, the destination Helper first warms up the TF session and waits for the checkpoint. The source Helper then asks TF to save the checkpoint, moves the checkpoint to destination in case of cross-server migration, and finally resumes the training session. To speed up the migration process, we adopt Ramdisk to keep the checkpoint in memory. In the cross-server case, the modified TF saves the checkpoint to the remote Ramdisk directly through the Network File System (NFS) protocol.

When the Migration Helper asks a job to perform checkpointing, the modified TF calls `tf.Saver` at the end of a mini-batch. For data parallelism, the checkpoint only includes the model in one GPU, regardless of the number of GPUs used in the training. To speedup TF migration further, we do not include the meta-graph structure in a checkpoint as it can be reconstructed based on user code.

In the warm-up phase, the modified TF checks the GPU configuration and reconstructs the meta-graph. It further creates the Executor to run a warm-up operation to ensure that the initialization is not deferred lazily. When resuming the training process, the modified TF loads the checkpoint, with multiple GPUs loading it in parallel, and continues the training.

6 Evaluation

In this section, we first present micro-benchmark results of the *Gandiva* mechanisms. We then evaluate the benefit *Gandiva* provides to multi-jobs. Finally, we present our evaluation results of the experiments on a 180-GPU cluster.

Our servers are 12-core Intel Xeon E5-2690@2.60GHz with 448GB RAM and two 40Gbps links (no RDMA), running Ubuntu 16.04. Each server has either four P100 or four P40 GPUs. All servers are connected to a network file-system called GlusterFS [3] with two-way replication on the server disks (SSDs). For jobs that use more than one GPU, we only evaluate data parallelism (as it is more common than model parallelism), and use synchronous updates (though we can support asynchronous update as well). Our evaluation uses 18 models, 8 implemented in PyTorch 0.3 and 10 implemented in TensorFlow 1.4. The batch size

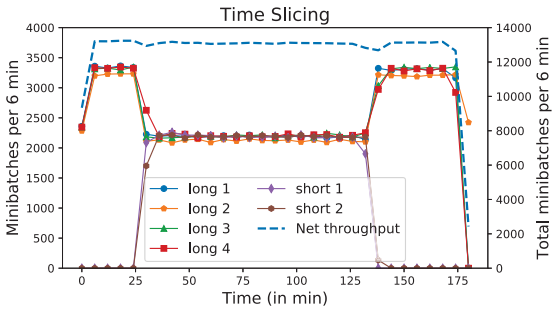


Figure 9: Time slicing six 1-GPU jobs on 4 GPUs.

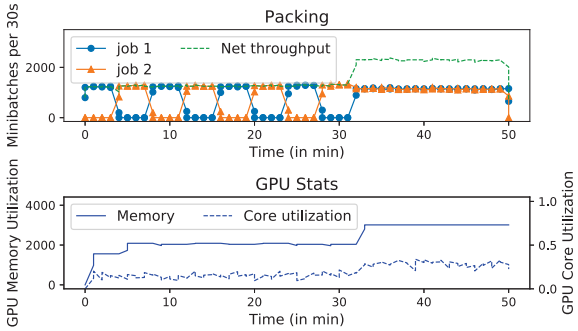


Figure 10: Packing jobs on single P40 GPU.

used for training are defaults from their references. All models take 6s or less per mini-batch in our evaluation. Thus, we set the time-slicing interval to 60s in these experiments.

6.1 Micro-benchmarks

In this section, we evaluate the *Gandiva* mechanisms, viz., time-slicing, packing, grow-shrink, and migration.

Time-slicing. We use six 1-GPU jobs on a single server with four P100 GPUs to illustrate time-slicing. These are ResNet-50 [24] models trained on the Cifar10 dataset using the PyTorch toolkit. When six 1-GPU jobs share four GPUs, each job ideally should get four minutes of GPU time out of every six.

Figure 9 shows a trace with the progress rate of each of the jobs over time. Initially, four 1-GPU long jobs are running and at time t=25min, two 1-GPU short jobs are scheduled at this server. One can see that the initial four 1-GPU jobs now get 4/6th their previous share. When the two short jobs depart, the long jobs return to their earlier performance. Also, note that the aggregate throughput of all jobs (right scale) is only marginally affected (less than 2%) during the entire trace, demonstrating that time-slicing is an efficient mechanism for providing early feedback during over-subscription.

Packing. Table 1 shows the performance of packing multiple jobs on a single GPU for various DLT mod-

Job	GPU Util (%)	Time Slicing (mb/s)	Packing Max (mb/s)	Packing Gain (%)
VAE [29]	8.7	81.8	419.3	412
SuperResolution [43]	14.1	40.3	145.2	260
RHN [58]	61.6	10.1	14.8	46
SCRNN [37]	66.8	16.7	23.3	39
MI-LSTM [52]	76.2	22.2	25.9	17
LSTM [5]	87.2	63.8	53.0	-16
ResNet-50 [24]	94.0	10.3	9.0	-13
ResNext-50 [53]	98.9	83.6	74.4	-11

Table 1: Packing multiple jobs on P40 (mb/s = minibatches/s).

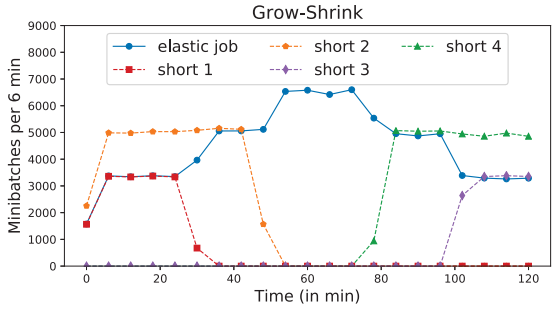


Figure 11: Grow from 1 to 4 GPUs, Shrink to 1-GPU.

els using PyTorch toolkit. For small DLT jobs with low GPU utilization, packing can provide significant gains of as much as 412%. For DLT jobs with middling GPU utilization, packing gains vary from model to model with some showing gains of up to 46%, but some exhibiting a loss of 16%. Finally, for image processing jobs with high utilization, such as ResNet-50 or ResNext-50 on the Cifar10 dataset, packing hurts performance by 11-13% .

Note that these packing results are without enabling NVIDIA's multi-process service (MPS) [7]. We found that MPS results in significant overhead in P40/P100 GPUs. However, hardware support for MPS in V100 GPUs [7] suggests that the use of MPS may be able to increase further packing gains in V100 GPUs.

Based on these results, predicting packing performance even with jobs of the same type appears challenging, let alone when jobs of different types are packed together. Instead, *Gandiva* adopts a profiling-based approach to packing. Figure 10 shows a case where two image super-resolution jobs [43] are initially being time-sliced on the same P40 GPU. After some time, the scheduler concludes that their memory and GPU core utilization is small enough that packing them is feasible and schedules them together on the GPU. The scheduler continues to profile their performance. Because their aggregate performance improves, packing is retained; otherwise (not shown), packing is undone and the jobs continue to use time-slicing.

Grow-Shrink. Grow-shrink is useful primarily when the cluster is under-utilized. *Gandiva* uses grow-shrink only for those jobs that specifically state that they can make use of this feature because users may want to adjust the learning rate and the batch size depending on the number of GPUs available. Figure 11 demonstrates this mechanism in action. Initially, a 4-P100 server has three jobs, 1-GPU growth-capable job, 1-GPU short job, and a 2-GPU short job, all using ResNet-50 with PyTorch. At time t=25min, the short job departs and after a time-out period of no new jobs being allocated to this GPU, the long running job expands to use 2 GPUs. At time t=45min, the second short job departs and the long running job expands to use all four GPUs. At time t=75min, a new 2-GPU job enters and the long job immediately shrinks to use two GPUs and, when another new 1-GPU job appears, the long job shrinks to use only 1 GPU. This micro-benchmark demonstrates that idle GPU resources can be effectively used with a mechanism like grow-shrink.

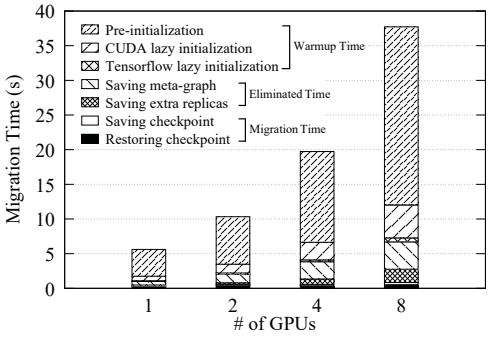


Figure 12: The breakdown of TF migration overhead.

Migration. We use a server with 8 P100 GPUs and the Tensorflow toolkit to evaluate migration overhead. First, we migrate a ResNet-50 single-server training job from one server to another. Figure 12 shows the detailed breakdown with a varying number of GPUs. Using our optimized implementation, we are able to eliminate or hide the majority of the migration overhead. The actual migration time, saving and restoring checkpoints, remains almost constant regardless of the number of GPUs because we save only one copy of the model. The loading of the in-memory checkpoint in each GPU runs in parallel and does not saturate the PCI-e bandwidth. The warm-up time and the cost due to meta-graph and checkpoints from other GPUs grow with the number of GPUs. As a result, we are able to save 98% of the migration overhead of 35s for 8-GPU jobs.

Figure 13 shows the max, min, and average intra-server and inter-server migration time of a 1-GPU job with 10 different deep learning models (summarized in Table 2) over 3 runs. Six of the 10 can be migrated within 1 second. Even the largest model (DeepSpeech [23] with a 1.4GB checkpoint) can be migrated in about 3.5 seconds, which is negligible compared to the long training time that often lasts for hours or days.

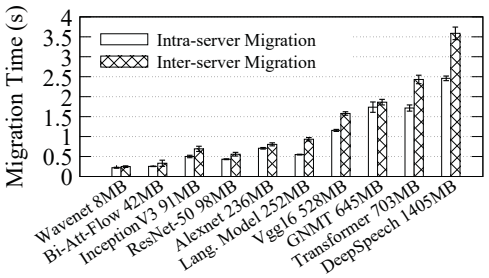


Figure 13: Migration time of real workloads.

6.2 Model exploration in a multi-job

AutoML, or automatic model exploration through hyper-parameter search is an important way to help users identify good neural models [21]. Typically, AutoML involves a hyper-parameter *configuration generator* and a *performance evaluator*. The generator uses different algorithms [11, 33] to generate new candidate configurations (DLT jobs), sometimes using the performance of prior configuration runs as a signal. The evaluator uses the early output of running jobs (*e.g.*, the learning curve) to predict the jobs' final performance and decide whether to continue running a given job or terminate it early.

Compared to a traditional scheduler where the number of configurations explored at any given time is limited by the number of GPUs available, *Gandiva* provides new primitives such as time-slicing and dynamic prioritization for AutoML algorithms to exploit. For example, the configuration generator is no longer limited by the number of GPUs and can dynamically generate many more configurations. Similarly, the performance evaluator can not only decide whether to continue or terminate a job but also how much priority to give to each configuration.

In this section, we explore one particular instance of using these new options enabled by *Gandiva* to highlight the potential benefit for AutoML. Detailed analysis of when and how many configurations to generate and/or how to best allocate priority among the various running configurations to utilize *Gandiva* features optimally is an open problem that we leave for future work.

At a high level, *Gandiva* can benefit an AutoML system in two ways. First, *Gandiva* can help AutoML explore more hyper-parameter configurations within a timespan, thereby enabling it to find better models [10, 19, 11]. Alternatively, *Gandiva* can help AutoML find a qualified model faster given a set of configurations through prioritization.

To demonstrate the benefit of *Gandiva* in exploring more configurations, we first use AutoML to run a multi-job to tune a LeNet-like CNN model with multiple convolution layers and fully connected layers, trained with the Cifar10 dataset. The hyper-parameters we search have 12 dimensions, including learning rate, dropout rate, number of layers, choice of optimization, etc. In this

	Neural model	Type	Dataset
10%	InceptionV3 [46]	CV	ImageNet [18]
	ResNet-50 [24]	CV	ImageNet
	Alexnet [31]	CV	ImageNet
	Vgg16 [44]	CV	ImageNet
60%	Bi-Att-Flow [42]	NLP	SQuAD [40]
	LanguageModel [56]	NLP	PTB [34]
	GNMT [51]	NLP	WMT16 [6]
	Transformer [49]	NLP	WMT16
30%	Wavenet [48]	Speech	VCTK [54]
	DeepSpeech [23]	Speech	CommonVoice [2]

Table 2: Neural models and the ratios in the trace.

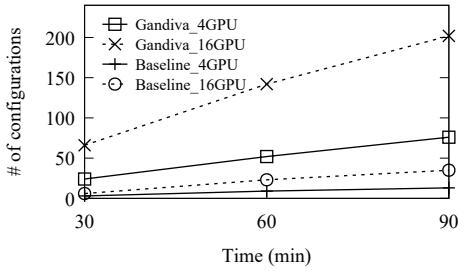


Figure 14: Model exploration number.

experiment, AutoML continually generates new hyper-parameter configurations based on Hyperopt [10] and leverages a curve-fitting method [19] to evaluate and predict the learning curve every 1,000 mini-batches (3% of the total mini-batches [19]). Jobs with no promise (less than 30% predicted accuracy) will be stopped early. The multi-job runs on 4 (or 16) P40 GPUs and each job requires 1 GPU. In the experiment, AutoML schedules 2 (or 8) more jobs every 1,000 mini-batches. In the baseline, jobs have to stay in a FIFO queue waiting for the running jobs to be terminated early or complete while in *Gandiva*, they are scheduled with time-slicing and migration support.

Figure 14 shows the number of explored hyper-parameter configurations. *Gandiva* can explore almost 10 times the number in the baseline approach in both the 4-GPU and 16-GPU cases. This is because, in the baseline approach, the GPUs can get "stuck" with a sub-optimal set of jobs that need to be run to completion, but in *Gandiva*, because of time-slicing, new configurations can be explored in parallel along with those jobs.

To demonstrate the benefit of *Gandiva* in finding a qualified model faster, we use Hyperopt to generate randomly the same set of 374 hyper-parameter configurations for both the baseline and *Gandiva*. The experiment measures the time required to find a configuration with at least 84% accuracy[2]. AutoML algorithms evaluate the jobs every 1,000 mini-batches and re-prioritize them based on the learning-curve prediction of their probability to achieve 84% accuracy [19]. In *Gandiva*, the top M jobs with the highest probabilities are then trained in the GPUs exclusively. In this experiment, we set M to 2 and 8 for 4-GPU and 16-GPU cases. Other jobs run in a time

[2]The LeNet-like CNN model is small: 84% is the best accuracy we found in the generated configurations.

	Position	93th (25%)	187th (50%)	280th (75%)	365th (98%)
4 GPUs	Baseline	691.5	1373.0	2067.2	2726.4
	Gandiva	125.5	213.8	302.4	387.1
	Speedup	5.51x	6.42x	6.84x	7.04x
16 GPUs	Baseline	253.0	492.7	731.7	970.0
	Gandiva	74.4	103.7	135.4	162.6
	Speedup	3.40x	4.75x	5.40x	5.96x

Table 3: Time to find a qualified configuration (minutes).

slicing manner. Our baseline approach stays the same as in the previous experiment. The result shows that *Gandiva* achieves 7x speedup compared to the baseline for the 4-GPU case and 6x for the 16-GPU case. More GPUs benefit the baseline as it implicitly improves the degree of parallelism of the long running jobs. There are two factors contributing to these gains. First, with prioritization, *Gandiva* grants more computation resources to the promising jobs. Second, because of the ability to run more configurations in parallel, *Gandiva* is able to find promising jobs quickly based on early feedback.

Further study shows the first job with the qualified configuration gets scheduled by *Gandiva* and the baseline in the 365^{th} place. We move the first qualified job from 365^{th} place to the first 25^{th} percentile, 50^{th} percentile, and 75^{th} percentile scheduling place and rerun the experiment. Table 3 summarizes the result: the later the qualified configuration shown, the larger gain *Gandiva* has. In a typical AutoML experiment, quality models usually show up later as those early-stopped jobs' configurations guide the system to find the better configurations.

To understand the sensitivity of *Gandiva*'s performance to the target accuracy of the model, we run AutoML with different target accuracies on a large state-of-the-art ResNet-like model (the official ResNet example in Keras [4]) for Cifar10. We use Hyperopt to generate 100 configurations, with the search space covering both the neural network architecture and various tunable hyper-parameters. The learning-curve prediction works as before; i.e., for every 3% of total mini-batches. The multi-job experiment runs on 16 P40 GPUs and every job runs on 1 GPU.

Table 4 shows the time spent on finding a model that is better than target accuracy using the baseline and *Gandiva* respectively. For a higher target accuracy, the performance gain of *Gandiva* is more notable. With 90% specified as a goal, the qualified model that is found achieves 92.62% validation accuracy. However, if the target accuracy is low; *e.g.*, 70%, a qualified model will appear early. In this case, the time for completing a single qualified configuration run dominates the total AutoML search time. Thus, *Gandiva* shows little benefit. We can see that when AutoML is used for achieving high accuracy models, *Gandiva* provides significant gains over the baseline.

Accuracy	70%	80%	90%
Baseline	134.1	2849.1	5296.7
Gandiva	134.1	543.1	935.4
Speedup	1.00x	5.25x	5.66x
Position	15th	58th	87th

Table 4: Model searching in ResNet-like network (minutes).

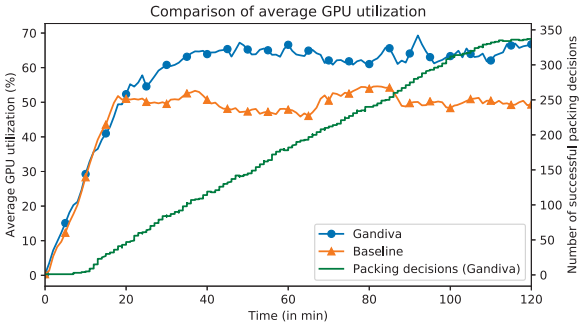


Figure 15: Cluster GPU utilization.

6.3 Cluster experiments: time-slicing and packing

In this section, we evaluate the *Gandiva* scheduler in a 45 server, 180-GPU cluster with about an equal mix of P100 and P40 GPUs. The scheduler implements both the reactive and introspective modes described earlier. In order to understand the gains contributed by different mechanisms in *Gandiva*, in this experiment, we only use time-slicing and packing, and disallow migrations. Further, none of the jobs are grow-shrink enabled. Thus, the accuracy achieved during training is unaffected by the *Gandiva* mechanisms.

We use the eight DLT jobs from Table 1 for this experiment and derive a mix of these jobs such that average GPU utilization is about 50%, similar to the average GPU utilization numbers reported from a study of a large deep learning cluster [28]. DLT jobs 1 and 2 from Table 1 (low utilization) are chosen with 0.3 probability, jobs 3, 4, and 5 (mid utilization) are chosen with 0.25 probability and jobs 6, 7, and 8 (high utilization) are chosen with probability of 0.45. Further, jobs 7 and 8 require either 2 or 4 GPUs while the rest each uses 1-GPU.

The number of mini-batches for each of these jobs are chosen such that, in isolation on P40, they take between 30 and 45 minutes of GPU time. A total of 1,000 jobs drawn from the above distribution arrive in a uniformly random manner over two hours. Using the same workload, we compare with *Gandiva* a baseline scheduler that does bin-packing but does not oversubscribe.

The primary goal of *Gandiva* is early feedback. We compute the average time to 100 mini-batches for all jobs as a measure of early feedback (*e.g.*, HyperBand [33] uses 100 mini-batches to evaluate a job). We find that the average time to complete 100 mini-batches is 498s for *Gandiva* and 2,203s for the baseline, for a reduction of 77%.

The second goal of *Gandiva* is cluster efficiency. Figure 15 shows the average GPU utilization of the cluster for the baseline scheduler and *Gandiva*, as well as the cumulative number of successful packings by *Gandiva* (right y-axis). The result shows clearly that *Gandiva* is able to use the cluster more efficiently than the baseline. The average utilization (computed over the stable regime from 20 to 200 mins) achieved by *Gandiva* is 62.8% compared to baseline average of 50.1%, resulting in a 26% relative improvement. Further, the greedy packing heuristic employed by *Gandiva* can be seen to be mostly successful with only a few packing decisions that need to be undone (the packing curve is mostly increasing with only small occasional dips).

6.4 Cluster experiments: time-slicing and migration

Trace. We collect a 9-day real-world job trace on a 2,000 GPU production cluster at Microsoft. The trace includes over 8,800 DLT jobs from three categories: computer vision (CV) (10%), Natural Language Processing (NLP) (60%), and Speech (30%), according to user survey and log analysis. However, the data/code used by these jobs are not available to us, due to security and privacy regulations. In their place, we pick 10 state-of-the-art deep learning models from Github with 50,000+ stars in total. The models are summarized in Table 2.

To synthesize a trace with similar characteristics as the production cluster, we mix these models with the same ratio as that in the trace. The number of mini-batches of the jobs in the trace are set to follow the job running time distribution of the 9-day real-world trace. We ensure that the synthesized trace closely follows the job running time distribution of the real-world trace, as shown in Figure 16. As before, none of the jobs are grow-shrink enabled in this experiment, as the cluster is in high load.

We run the trace using Hadoop's YARN capacity scheduler [50] and our *Gandiva* scheduler.

Fast-forwarding. To speed up replaying the 9-day trace, we leverage the predictability of the 10 models. We use the scheduler to instruct a running job to skip a number of mini-batches (i.e., fast-forwarding) whenever there are no scheduling events, including job arrival, departure, and migration, etc. The time skipped is calculated by measuring the previous mini-batch performance when the job reaches a stable state.

We validate fast-forwarding by constructing a 3-hour trace and compute average job completion time (JCT) and the makespan (the running time for the entire experiment) for the full trace and the experiment with fast-forwarding enabled using the capacity scheduler and *Gandiva*. The difference between the real and fast-forwarded experiment in all cases was less than 1%.

	Avg. JCT (mins)	Makespan (mins)
Cap. Sche.	832	13371
Gandiva	656	11349
Improvement	26.8%	17.8%

Table 5: Full trace experiment with fast-forwarding

Table 5 shows the average job completion time and the makespan for the two schedulers when replaying the synthesized job trace in a cluster with 100 GPUs (50 P100, 50 P40). We see that *Gandiva* improves average JCT by 26.8% and the total makespan is reduced by 17.8%. Figure 17 shows the CDF of the JCT of the two approaches: it shows *Gandiva* has more jobs with a JCT less than around 100 mins. During the entire experiment, *Gandiva* initiates migration 470 times; i.e., approximately once every 20 minutes.

Multi-job performance in a shared cluster. To compare the AutoML performance of a multi-job in a shared environment, we run the synthesized trace the same way as earlier in the same cluster with 100 GPUs. The trace-driven jobs act as background jobs, emulating a realistic shared cluster environment. At the $5,607^{th}$ minute (roughly in the middle of the trace), we launch two multi-jobs, each to find a qualified CNN model described in Section 6.2, trained on the Cifar10 dataset. Each multi-job is allocated 8 GPUs. For fair comparison, each multi-job is allowed to preempt other jobs to get 8 GPUs to reduce the unpredictable resource sharing.

We are particularly interested in understanding the effect of migration in *Gandiva* and therefore use a 2-GPU VGG-like model that is large and locality sensitive (Section 3.1). Each AutoML job runs for 100,000 mini-batches and reports the learning curve every 3,000 mini-batches (3%). Like the previous experiment, the job can be early stopped if the learning shows no promise [19]. In this experiment, the AutoML algorithm tunes the learning rate of the model with 40 configurations. The multi-job completes if a job's model achieves 99.5% training accuracy, with 91.3% validation accuracy. Again, the top M highest probability jobs run exclusively while other jobs are time-sliced. In this experiment, we set M to 2 (i.e., 4 GPUs).

As shown in Figure 18, with the capacity scheduler, it takes 1,215.74 and 1,110.62 mins, respectively, to find the qualified configuration for the two multi-jobs. *Gandiva*'s mechanisms like migration, time-slicing, and dynamic priority help provide better locality, identify promising jobs earlier, and improve the training speed of high priority jobs. As a result, *Gandiva* achieves a speedup of 13.6 and 12.9, respectively. Based on a micro-benchmark we did, we observed that time-slicing alone gave 7x gains for this AutoML experiment. Thus, the rest of the gains are attributable to improved locality due to migration. A real example of migration observed in this experiment was shown in Figure 8.

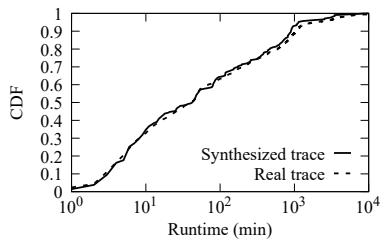


Figure 16: CDF of the synthesized trace

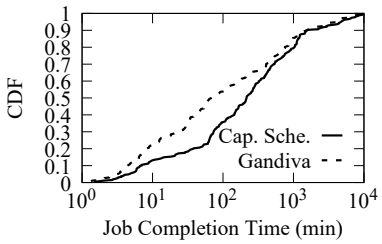


Figure 17: CDF of job completion time

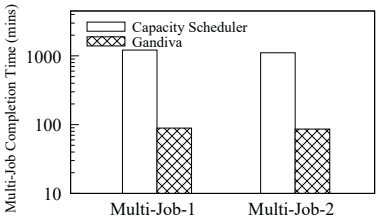


Figure 18: Multi-job completion time

7 Related Work

DLT job scheduling today. DLT jobs are scheduled today by big data schedulers such as Kubernetes or YARN [12, 28]. In these systems, a *fixed set of GPUs is assigned exclusively for the lifetime of a DLT job*. Thus, job queueing times can vary from a few minutes [12] to even hundreds of minutes [28] in these clusters.

An earlier study [28] shows that the average GPU utilization in a production cluster was only around 52%. Some jobs can inherently result in low GPU utilization due to the use of small models [29] and/or the use of small batch sizes for better generalization [35]. Further, jobs with inherently high GPU utilization can be adversely affected by poor GPU affinity and/or interference.

Scheduling policies for machine learning. Recent research [9, 15] also suggests that locality, interference, and GPU utilization are important performance factors for GPU workloads. They develop analytical models to predict the performance of GPU workloads. A *Gandiva* scheduler may leverage such models to guide its scheduling decisions. At its core, *Gandiva* framework is designed to empower DLT schedulers with the primitives such as time-slicing and migration.

SLAQ [57] proposes a scheduling policy that prioritizes resources in a CPU-based cluster to Spark jobs with high potentials (*e.g.*, the one with a fast improving learning curve). *Gandiva* can leverage the same policy for DLT on GPU clusters. Optimus [39] derives a proper number of parameter-servers and workers for MxNet-based deep learning jobs, which complements *Gandiva* in GPU cluster scheduling.

AutoML. *Gandiva* enables the co-design of DLT schedulers and AutoML algorithms like [10, 30]. Jobs in a multi-job can be promoted dynamically with more resource and/or better locality, accordingly to AutoML specific algorithms. Google Vizier [21], HyperDrive [41], and TuPAQ [45] focus more on the system design of AutoML. *Gandiva* empowers these systems with lower level system primitives that can further improve AutoML training experience in a multi-job, as shown in the experiments.

Big data cluster scheduling frameworks. Most recent big data scheduling frameworks assume jobs are modeled after a data flow graph (DFG) [26, 55, 13, 27, 20, 22]. Map/Reduce like tasks instantiated from the logical DFG get scheduled dynamically according to the job progress and the DFG dependency. *Gandiva* instead relies on the micro-task boundary implicitly defined by the mini-batch boundary. The low-level mechanisms of *Gandiva* such as time-slicing and migration also differ significantly from those big data scheduling systems [13, 22, 25, 14], while being surprisingly similar to a traditional operation system [47].

Time-slicing, suspend-resume, and process migration. *Gandiva* adopts traditional OS process primitives to facilitate DLT scheduling [47]. Unlike the general purpose OS mechanisms, *Gandiva* leverages the intra-job predictability of DLT to achieve a highly efficient implementation. *Gandiva* does not claim generality of the proposed techniques to other application domains.

8 Conclusion

We present *Gandiva*, a cluster scheduling framework for deep learning, which provides a set of efficient, low-level system primitives such as time-slicing, migration, intra-job elasticity, and dynamic priority. Using these primitives, *Gandiva* can effectively support neural model exploration in a multi-job, finding accurate neural models up to an order of magnitude faster than using traditional schedulers in a realistic shared cluster environment. *Gandiva* provides an efficient implementation of the proposed mechanisms by exploiting the intra-job predictability of DLT: our system prototype demonstrates that job suspend/resume and migration can be achieved under a second, even for cross-server migration for popular deep learning toolkits such as Tensorflow and PyTorch. Combined with an introspective scheduling policy, *Gandiva* improves overall cluster utilization by 26%.

Acknowledgments

We thank our shepherd KyoungSoo Park and the anonymous reviewers for their valuable comments and suggestions. We thank Bin Wang and Shuguang Liu from Bing search platform team and Daniel Li, Subir Sidhu, and Chandu Thekkath from Microsoft AI Platform team for providing access to the GPU clusters and Azure GPU VMs.

References

[1] Checkpoint/restore in user space. https://criu.org/Main_Page.

[2] Common voice dataset. https://voice.mozilla.org/.

[3] GlusterFS. https://docs.gluster.org/en/latest/.

[4] Keras. https://github.com/keras-team/keras.

[5] LSTM training on wikitext-2 dataset. https://github.com/pytorch/examples/tree/master/word_language_model.

[6] WMT16 dataset. http://www.statmt.org/wmt16/.

[7] Multi-process service, Oct 2017. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[8] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), vol. 16, USENIX Association, pp. 265–283.

[9] AMARAL, M., POLO, J., CARRERA, D., SEELAM, S., AND STEINDER, M. Topology-aware GPU scheduling for learning workloads in cloud environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), ACM, p. 17.

[10] BERGSTRA, J., KOMER, B., ELIASMITH, C., YAMINS, D., AND COX, D. D. Hyperopt: a Python library for model selection and hyperparameter optimization. *Computational Science & Discovery 8*, 1 (2015), 014008.

[11] BERGSTRA, J. S., BARDENET, R., BENGIO, Y., AND KÉGL, B. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 2546–2554.

[12] BOAG, S., DUBE, P., HERTA, B., HUMMER, W., ISHAKIAN, V., JAYARAM, K., KALANTAR, M., MUTHUSAMY, V., NAGPURKAR, P., AND ROSENBERG, F. Scalable multi-framework multi-tenant lifecycle management of deep learning training jobs. In *Workshop on ML Systems, NIPS* (2017).

[13] BOUTIN, E., EKANAYAKE, J., LIN, W., SHI, B., ZHOU, J., QIAN, Z., WU, M., AND ZHOU, L. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 285–300.

[14] BURNS, B., GRANT, B., OPPENHEIMER, D., BREWER, E., AND WILKES, J. Borg, omega, and kubernetes. *ACM Queue 14* (2016), 70–93.

[15] CHEN, Q., YANG, H., GUO, M., KANNAN, R. S., MARS, J., AND TANG, L. Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2017), ASPLOS '17, ACM, pp. 17–32.

[16] CHEN, T., XU, B., ZHANG, C., AND GUESTRIN, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[17] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM 51*, 1 (Jan. 2008), 107–113.

[18] DENG, J., DONG, W., SOCHER, R., LI, L.-J., LI, K., AND FEI-FEI, L. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on* (2009), IEEE, pp. 248–255.

[19] DOMHAN, T., SPRINGENBERG, J. T., AND HUTTER, F. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI* (2015), vol. 15, pp. 3460–8.

[20] GOG, I., SCHWARZKOPF, M., GLEAVE, A., WATSON, R. N. M., AND HAND, S. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 99–115.

[21] GOLOVIN, D., SOLNIK, B., MOITRA, S., KOCHANSKI, G., KARRO, J., AND SCULLEY, D. Google Vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017), KDD '17, pp. 1487–1495.

[22] GRANDL, R., KANDULA, S., RAO, S., AKELLA, A., AND KULKARNI, J. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 81–97.

[23] HANNUN, A., CASE, C., CASPER, J., CATANZARO, B., DIAMOS, G., ELSEN, E., PRENGER, R., SATHEESH, S., SENGUPTA, S., COATES, A., ET AL. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* (2014).

[24] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[25] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R. H., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI* (2011), vol. 11, pp. 22–22.

[26] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 59–72.

[27] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 261–276.

[28] JEON, M., VENKATARAMAN, S., QIAN, J., PHANISHAYEE, A., XIAO, W., AND YANG, F. Multi-tenant GPU clusters for deep learning workloads: Analysis and implications. *MSR-TR-2018-13* (2018).

[29] KINGMA, D. P., AND WELLING, M. Stochastic gradient VB and the variational auto-encoder. In *Second International Conference on Learning Representations, ICLR* (2014).

[30] KLEIN, A., FALKNER, S., SPRINGENBERG, J. T., AND HUTTER, F. Learning curve prediction with bayesian neural networks.

[31] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[32] LECUN, Y., BENGIO, Y., AND HINTON, G. Deep learning. *nature 521*, 7553 (2015), 436.

[33] LI, L., JAMIESON, K., DESALVO, G., ROSTAMIZADEH, A., AND TALWALKAR, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560* (2016).

[34] MARCUS, M. P., MARCINKIEWICZ, M. A., AND SANTORINI, B. Building a large annotated corpus of English: The penn treebank. *Computational linguistics 19*, 2 (1993), 313–330.

[35] MASTERS, D., AND LUSCHI, C. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612* (2018).

[36] MIIKKULAINEN, R., LIANG, J., MEYERSON, E., RAWAL, A., FINK, D., FRANCON, O., RAJU, B., SHAHRZAD, H., NAVRUZYAN, A., DUFFY, N., ET AL. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548* (2017).

[37] MIKOLOV, T., JOULIN, A., CHOPRA, S., MATHIEU, M., AND RANZATO, M. Learning longer memory in recurrent neural networks. *arXiv preprint arXiv:1412.7753* (2014).

[38] PASZKE, A., GROSS, S., CHINTALA, S., AND CHANAN, G. PyTorch, 2017.

[39] PENG, Y., BAO, Y., CHEN, Y., WU, C., AND GUO, C. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth European Conference on Computer Systems* (2018), ACM.

[40] RAJPURKAR, P., ZHANG, J., LOPYREV, K., AND LIANG, P. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[41] RASLEY, J., HE, Y., YAN, F., RUWASE, O., AND FONSECA, R. HyperDrive: Exploring hyperparameters with POP scheduling. In *Proceedings of the 18th International Middleware Conference, Middleware* (2017), vol. 17.

[42] SEO, M., KEMBHAVI, A., FARHADI, A., AND HAJISHIRZI, H. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603* (2016).

[43] SHI, W., CABALLERO, J., HUSZÁR, F., TOTZ, J., AITKEN, A. P., BISHOP, R., RUECKERT, D., AND WANG, Z. Real-time single image and video super-resolution using an efficient subpixel convolutional neural network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 1874–1883.

[44] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[45] SPARKS, E. R., TALWALKAR, A., HAAS, D., FRANKLIN, M. J., JORDAN, M. I., AND KRASKA, T. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 368–380.

[46] SZEGEDY, C., VANHOUCKE, V., IOFFE, S., SHLENS, J., AND WOJNA, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 2818–2826.

[47] TANENBAUM, A. *Distributed operating systems*. Prentice Hall, 1995.

[48] VAN DEN OORD, A., DIELEMAN, S., ZEN, H., SIMONYAN, K., VINYALS, O., GRAVES, A., KALCHBRENNER, N., SENIOR, A., AND KAVUKCUOGLU, K. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499* (2016).

[49] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, Ł., AND POLOSUKHIN, I. Attention is all you need. In *Advances in Neural Information Processing Systems* (2017), pp. 6000–6010.

[50] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., ET AL. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), ACM, p. 5.

[51] WU, Y., SCHUSTER, M., CHEN, Z., LE, Q. V., NOROUZI, M., MACHEREY, W., KRIKUN, M., CAO, Y., GAO, Q., MACHEREY, K., ET AL. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[52] WU, Y., ZHANG, S., ZHANG, Y., BENGIO, Y., AND SALAKHUTDINOV, R. R. On multiplicative integration with recurrent neural networks. In *Advances in Neural Information Processing Systems* (2016), pp. 2856–2864.

[53] XIE, S., GIRSHICK, R., DOLLÁR, P., TU, Z., AND HE, K. Aggregated residual transformations for deep neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on* (2017), IEEE, pp. 5987–5995.

[54] YAMAGISHI, J. English multi-speaker corpus for CSTR voice cloning toolkit, 2012. *URL http://homepages. inf. ed. ac. uk/jyamagis/page3/page58/page58. html*.

[55] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, ., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'08: Eighth Symposium on Operating System Design and Implementation* (December 2008), USENIX.

[56] ZAREMBA, W., SUTSKEVER, I., AND VINYALS, O. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[57] ZHANG, H., STAFMAN, L., OR, A., AND FREEDMAN, M. J. SLAQ: Quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), SoCC '17, pp. 390–404.

[58] ZILLY, J. G., SRIVASTAVA, R. K., KOUTNIK, J., AND SCHMIDHUBER, J. Recurrent highway networks. In *International Conference on Machine Learning* (2017), pp. 4189–4198.