DynamoML: Dynamic Resource Management Operators for Machine Learning Workloads

Min-Chi Chiang and Jerry Chou
National Tsing Hua University,Hsinchu, Taiwan
louisby0123@gmail.com, jchou@lsalab.cs.nthu.edu.tw

Keywords: Deep learning, GPU resource management, Job scheduling, Performance optimization

Abstract: The recent success of deep learning applications is driven by the computing power of GPUs. However, as the workflow of deep learning becomes increasingly complicated and resource-intensive, how to manage the expensive GPU resources for Machine Learning (ML) workload becomes a critical problem. Existing resource managers mostly only focus on a single specific type of workload, like batch processing or web services, and lacks runtime optimization and application performance awareness. Therefore, this paper proposes a set of runtime dynamic management techniques (including auto-scaling, job preemption, workload-aware scheduling, and elastic GPU sharing) to handle a mixture of ML workloads consisting of modeling, training, and inference jobs. Our proposed system is implemented as a set of extended operators on Kubernetes and has the strength of complete transparency and compatibility to the application code as well as the deep learning frameworks. Our experiments conducted on AWS GPU clusters prove our approach can out-perform the native Kubernetes by 60% system throughput improvement, 70% training time reduction without causing any SLA violations on inference services.

1 Introduction

Deep Learning (DL) is popular in data-center as an important workload for artificial intelligence, because it powers variety of applications, including image classification (He et al., 2016; Krizhevsky et al., 2017), object detection (Redmon et al., 2015; Xu et al., 2017), language processing (Vaswani et al., 2017; Devlin et al., 2018; Yang et al., 2019; Liu et al., 2019; Lan et al., 2019) to self-driving cars (Tian et al., 2017) and autonomous robotics (Levine et al., 2016). However, deep learning is also known to be computing intensive. As reported in a recent survey (Amodei and Hernandez, 2018), the amount of computations used in the largest AI training runs has been increasing exponentially with a 3.4-month doubling time, which is at the pace even faster than the Moore's Law. The increasingly popular trend of AutoML techniques, such as automatic hyper-parameter tuning and network architecture search, further pushes the need of computing power as models must be repeatedly trained with different settings in order to refine DL models.

GPUs have emerged as a popular choice for deep learning applications because of their tremendous throughput powered by massive parallelism. Hence, today's deep learning production systems are mostly built on shared multi-tenant GPU clusters where abundant computing resources can be utilized and shared among users to enable large-scaled model training, and highly efficient model inference serving. Therefore, it has drawn increasing attention from both industry and research communities to improve the efficient and performance of expensive GPU resources for DL workloads.

Resource management (such as resource allocation, job scheduling) is one of main approaches for improving job performance, system throughput and hardware utilization. But managing the resources of DL workload on GPU cluster can be challenging because DL production needs several processing stages from data pre-processing to model training, validation, and finally model deployment for serving inferences. The workflow is also known as the ***ML pipeline***. For simplicity, we refer the computation before model training as the ***modeling jobs***; the computations for modeling and hyper-parameter tuning as the ***training jobs***; and the computations for serving model inference requests as the ***inference jobs***. The workload characteristics of each of these jobs can be widely different from each other. For instance, modeling jobs can be non-GPU bound jobs because

some of the data pre-processing tasks involve interactive data analysis, and others may hardly be paralleled using GPU, such as feature extraction in advertisement (He et al., 2014) and data augmentation in computer vision (Zoph et al., 2019). Training and inference jobs both require GPUs, but training behaves like parallel batch processing with huge and stable GPU usage pattern, while inference behaves like web service with short and bursty GPU usage pattern. In addition, inference jobs can have strict SLA requirements on the request response time. In contrast, training jobs can be suspended through the checkpoint-restart mechanism. The mixture and diverse computing jobs in DL workload creates difficulties as well as opportunities on resource management.

Existing cluster schedulers (e.g., Borg (Verma et al., 2015), YARN (Vavilapalli et al., 2013)) are designed for general-purpose workload. Their scheduling algorithms (e.g., DRF (Ghodsi et al., 2011), TetriSched (Tumanov et al., 2016), corral (Jalaparti et al., 2015), HT-Condor (Tannenbaum et al., 2001)) are mostly designed to ensure resource fairness and utilization by allocating a fixed amount of resources for each job according to the resource requirements specified by the job owner upon job submission. As a result, existing approach may lead to sub-optimal application performance and system throughput due to the following reasons.

1. **Application-oblivious scheduling:** Existing schedulers only concern about the amount of resources allocated to each job without being aware of the application performance and resource characteristics. For training jobs, the location of allocated GPUs can significantly affect training time due to communication overhead. For inference jobs, their performance is measured by SLA guarantee which depends not only on the resource allocations, but also on the time-varied service workload (i.e., inference requests from clients). For modeling jobs, they are consisted of non-GPU and interactive workload, so their performance can be much less sensitive to GPU resources. Hence, without properly considering the performance impact from resource allocations can easily lead to under-utilization or under-provisioning problems.
2. **Static resource management:** Due to the workload diversity and time-varied resource demands of DL workload, dynamic resource management like job preemption, auto-scaling, is essential to guarantee application performance and resource utilization. But existing cluster managers often require job owners to manually operate or resubmit their jobs in order to adjust the resources of jobs. The lack of resource management experience and information further discourage users to adjust their job resources when necessary.
3. **Coarse-grained GPU allocation:** Due to the lack of multi-tasking management of GPU device, the minimum granularity of GPU allocation today is a single GPU device. That means an application can have multiple GPUs, but each GPU can only be allocated to exactly one application. While DL jobs can be accelerated using GPUs, a single DL job may not always utilize the whole GPU card due to reasons like memory-bound training jobs with large batch sizes or large network models, non GPU-bound modeling jobs with human interactions, and inference job with time-varied workloads. Hence, existing GPU cluster without supporting GPU sharing can result in low resource utilization and system throughput.
4. **Homogeneous workload consideration:** Not until recently, DL domain specific management systems (Peng et al., 2018; Xiao et al., 2018; Xiao et al., 2020) have been proposed to tackle the issues above. However, all of them focus on DL training jobs only. With the emerging trend of MLOps and AutoML that requires to unify ML system development (Dev) and ML system operation (Ops) together, end-to-end ML pipelines with mixture workloads are likely to be ran and managed in a shared resource pool provided by a single production system. Therefore, resource manager should be designed with the consideration of jobs with different performance metrics, workload characteristics, and execution priorities.

To address the aforementioned problems, we present ***DynamoML***, a Dynamic resource Management Operators for Machine Learning workloads. *DynamoML* consists of three modualized runtime resource management operators. First operator manages fine-grained shared GPU allocation to increase GPU utilization. Second operator is an application-aware scheduler to improves the performance of distributed model training. Third operator is a performance-driven auto-scaling controller to guarantees the SLA requirement of model inference services. The three operators can further collaborate to each other to provide dynamic resource allocation across different types of jobs through preemptive and priority scheduling. Our system is designed and implemented as the extended framework components to the Kubernetes, which has become the de facto resource manager of containerized clusters and data centers. Our experiments running a mixture of ML pipeline workload on a 16-GPUs cluster show that *DynamoML* improves the native Kubernetes by

increasing the system throughput by more than 60%, reducing the average training time by 70%, and eliminating all the SLA violations. Our in-depth analysis also show that without proper coordination and collaboration between different management techniques to balance the resource between training job and inference jobs, several issues could occur, including SLA violation on inference service, wasted idle GPUs, prolonged model training time.

The rest of the paper is structured as follows. Section 2 discusses the DL workload characteristics and overall system architecture. Section 3 describes the design and implementation of each DynamoML operators. Section 4, and Section 5 presents our experiment setup and results, respectively. Finally, the related work discussion is in Section 6, and the paper is concluded in Section 7.

2 System and Workload Model

2.1 ML Pipeline Workload

In this work, we consider a GPU cluster running a set of computing jobs produced from the ML pipeline workflow. In general, the jobs can be classified into three types: modeling, training, and inference. The workload characteristics of these three types of jobs are summarized in Table 1, and briefly discussed as follows.

- Modeling: We use modeling jobs to represent all the computing jobs before model training. In general, modeling jobs involve neural network model building and interactive data pre-processing, including data cleaning, labeling, validations, and feature extraction. In practice, users normally perform these tasks through web notebooks, such as Juypter Notebook. Since the computations in this pipeline stage commonly involves data processing and human interaction, the GPU usage is low, some execution delay can be tolerated, and the workload pattern can be bursty.
- Training: Model training can be extremely time consuming. In order to reduce training time, distributed model training across multiple GPU-nodes has been supported by the mainstream deep learning frameworks, such as Tensorflow, PyTorch, and Keras. Most of them adapt the BSP (Bulk Synchronous Parallel) computing method to implement the data parallel model training, where a training job is consisted of a set of worker processes, and all the workers must synchronously aggregate their gradients at the end of each training iteration in order to update the model weights. Since training time can be long, modern deep learning libraries also support checkpoint mechanism to tolerant faults and to restart training with different resource configurations. Therefore, comparing other two types of jobs, training has the highest resource usage and the lowest urgency.
- Inference: To serve model's inference requests from clients, an inference job often packages the model and deploys it as a web service (e.g., TF-Server). Similar to web services, the workload of a web service can be time-varied according to the number of client requests, and the SLA requirement of a service can be guaranteed. Therefore, inference jobs should have the highest urgency with periodic workload pattern and medium GPU usage demand.

2.2 DynamoML Operators

According to the aforementioned workload characteristics, we proposed and implemented a resource management framework called DynamoML. The goal of our system is to support efficient ML pipeline workload executions on GPU cluster by maximizing resource utilization while satisfying application performance requirements.

Our work is implemented as an extended framework on Kubernetes, because Kubernetes has become the most popular resource orchestrator for hosting containerized computing workload. However, Kubernetes, like our cluster schedulers (e.g., Borg (Verma et al., 2015), YARN (Vavilapalli et al., 2013), HTCondor (Tannenbaum et al., 2001)), is lack of proper resource management for DL workloads. Our resource management techniques are implemented as three modualized operators:

- Shared GPU allocator: it enables fine-grained shared GPU allocation in Kubernetes. Hence, the GPU utilization can be increased by allowing multiple non-GPU bound modeling jobs to share a single GPU. In comparison, the native device plug-in framework of Kubernetes doesn't allow fractional allocation.
- Distributed training job scheduler: It is a runtime scheduler that addresses several resource allocation problems of distributed model training jobs. (1) It reduces the communication overhead of distributed training by packing the workers of a training job on a single node . (2) It avoids the idle resource problem of synchronous computations by proving gang scheduling, so that all the workers of a training job will be scheduled together as a

Table 1: Summary of ML Pipeline workload characteristics and their corresponding resource management solutions.

Job Type	Workload Characteristics				Problems	Solutions	Results
	type	pattern	usage	urgency			
Modeling (e.g., Notebook)	interactive data analysis	Bursty	Low	Med	1.Low utilization	Fractional & elastic GPU allocation	Increase utilization
Training (e.g., TFJob)	BSP (Bulk Synchronous Parallel)	Persistent	High	Low	1.Resource monopoly 2.Communication 3.Synchronization	1.Gang & locality aware scheduling 2.Task preemption	1.Avoid idle resources 2.Reduce training time
Inference (e.g., TFServer)	Web service	Periodic	Med	High	1.SLA requirement 2.Elastic workload	Auto-scaling	Avoid SLA violation

group. (3) It uses checkpoint mechanism to force resource preemption on training jobs, so the resources won't be monopolized by the long running training jobs. In comparison, the native Kubernetes scheduler cannot achieve these goals because it lacks of the awareness of application performance.

- Inference service auto-scaling controller: It aims to dynamic add or remove the server instances of a service job according to the response time of inference requests, so that an application level SLA requirement can be guaranteed under workload variations. In comparison, the existing scaling mechanism in Kubernetes is based on resource usage not application performance. Furthermore, when the system is lack of resources for inference jobs, auto-scaling controller can ask job scheduler to release the resources of training jobs. Therefore, inference jobs always have the highest scheduling priority in our framework.

2.3 Design Requirements & Strengths

Besides the goal of resource management, DynamoML also has the following strengths from its design requirements.

- Transparency: All the resource control mechanism of DynamoML are implemented as the extended components of Kubernetes using the technique like custom controller, sidecar container, and library hooks. Therefore, no code modification to the deep learning computing frameworks or user program. Because of transparency, our system can work seamlessly with Kubernetes, and our resource management strategy can also be applied any application with the same targeted workload characteristics.
- Modularization: each component of DynamoML provides a standalone management service, such as GPU sharing, scheduling and auto-scaling. These services are triggered by their service-defined API and event. Hence, system administrators can independently deploy the individual components of DynamoML according to their needs.
- Agility: To overcome the workload variation and diversity of ML pipeline jobs, DynamoML focuses on runtime resource sharing and management. Form resource allocation aspect, our GPU sharing supports *elastic allocation* which allows the actual resource usage to be bounded between a user specified range specified by a pair of values (request, limit), so that the resource within the range can be elastically shared among users. From resource usage aspect, DynamoML supports auto-scaling and preemption on training and inference jobs, so that resource demands can be adjusted and adapted to the runtime application behavior and performance.

3 System Implementation

3.1 Shared GPU Allocator

GPU sharing is a necessary mean to improve resource utilization, especially for modeling and inference jobs. Our GPU sharing solution consists of two parts. First is to enable fractional GPU allocation in Kubernetes. Second is to ensure the GPU resource can be shared fairly among containers.

The device plug-in framework of Kubernetes treats GPU device as a single non-divisible resource object, so fractional GPU allocation is not allowed. To overcome this limitation, our system first launches a set of pods to allocate the GPU resources from Kubernetes, and obtain GPU devices' UUID. Then the same GPU can be attached to multiple containers installed with nvidia docker package by setting the GPU's UUID in in the environment variable "NVIDIA_VISIBLE_DEVICES". To support fractional allocation, our shared GPU allocator will track the residual resource amount on each GPU, and ensure GPUs are not over-allocated.

After a GPU is attach and accessible by a container, we still have to ensure the actual resource us-

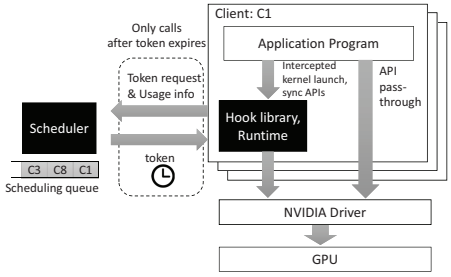

Figure 1: GPU usage control framework.

age doesn't exceed the allocation amount. As shown in Figure 1, to throttle the GPU usage of a container, we insert a LD_PRELOAD hook library in the container to intercept its GPU API. The intercepted GPU APIs will be blocked until the hook library receives an execution token from the scheduler of our GPU allocator. A time-sharing scheduler is implemented to pass the token around containers according to their resource allocation demand. Therefore, the resource usage of a container cannot exceed its allocated demand. To maximize GPU utilization, we further support elastic allocation, which allow users to specify their minimum and maximum demand as (request, limit), so that the requested resource can be reserved and guaranteed, while the residual capacity can still be utilized by the container without exceeding its limit. More details of the design and implementation can be found in our previous work (Yeh et al., 2020).

3.2 Training Job Scheduler

The native Kubernetes scheduler schedules the container (i.e. pod) of each worker task independently on random nodes with sufficient resources. However, distributed training jobs are communication-bound with task dependency among each other. So the naive FCFS random scheduling algorithm of native Kubernetes scheduler can cause significant communication and synchronization overhead.

To address the above issues, we developed our own job-level scheduler on top of the task-level native scheduler to schedule and manage all the tasks of a training job as a unit. Through our scheduler, a job is only launched when the system has enough residual capacity to run all its tasks simultaneously. Thus, resources will not be occupied by tasks waiting for synchronization. To minimize the communication overhead of a job, our scheduler tends to pack all the tasks of a job on as fewer number of compute nodes as possible. The location of the task is controlled by specifying the "node selector" label in the pod spec of the workers, so the native scheduler can only create the pods on the designated nodes of our scheduler.

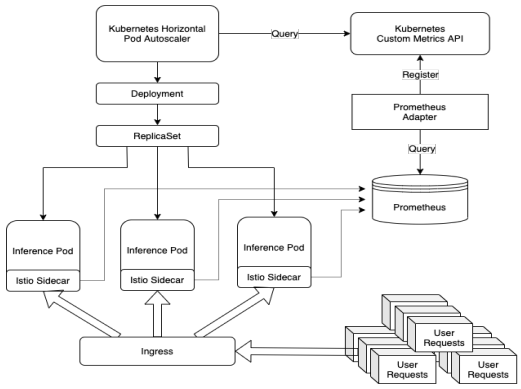

Figure 2: The system design diagram of the inference auto-scaling controller.

Finally, our scheduler monitors the system resource usage status to dynamically adjust the number of workers of training jobs. Additional workers are added to training jobs when the system loading is low, so that jobs can take advantage of the residual capacity to reduce their execution time. On the other hand, workers can also be taken from jobs when the system loading is high, so that the resources can be reclaimed from the running jobs to launch the waiting jobs as soon as possible. More details of the design and implementation can be found in our previous work (Lin et al., 2019).

3.3 Inference Auto-scaling Controller

In order to dynamically scale up and down in response to a varied number of users' requests, we integrated Kubernetes Horizontal Pod Autoscaler (HPA). However, the naïve Kubernetes Horizontal Pod Autoscaler (HPA) is limited to scale pods according to the current state of CPU or memory consumption. The most direct way to scale pods should depend on the current number of users' requests, which requires us to integrate third-party solutions to fulfill this goal. We further integrate Istio, a popular tool to build service mesh in our implementation, to gather the requests' information within the cluster.

The term service mesh is used to describe the network of microservices that make up such applications and their interactions. Its requirements can include service discovery, load balancing, metrics, and monitoring. Istio's solution creates a network of deployed services with traffic management, security, observability, and extensibility. It directly attaches a sidecar, a proxy to help pod exchanging information, to every pod. Whenever there exists a need to pass data between pods, the sidecar would intercept all network communication. By intercepting the communications, Istio generates metrics for all service traf-

fic in, out, and within an Istio service mesh. These metrics provide information on behaviors such as the overall traffic volume, the error rates within the traffic, and the response times for requests. As a result, we use Istio as our primary tool to gather the communications metrics.

As shown in Figure 2, when we successfully fetch the metrics from Istio, we need to store the information for Kubernetes Custom Metrics API to query. Although we're able to forward data from Istio to Kubernetes Custom Metrics API directly, we could further store, query, and monitor the metrics if we store in Prometheus. By employing Prometheus, a popular Time-Series Database, to periodically forward information to Prometheus, we gather the metrics, including the request number per second and response time. Then, since Kubernetes HPA is only able to fetch metrics from Kubernetes Custom Metrics API, we install the Prometheus Adapter and register it to Kubernetes Custom Metrics API. Finally, Kubernetes HPA is capable of scale up and down pods according to the current number of users' requests.

In order to prioritize the inference jobs and preempt the training jobs if needed. Whenever there's a necessity to increase the inference pods in response to flooded users' requests, our auto-scaling controller would check whether current residual resources are available for inference jobs to scale up. If not, we would sequentially evict the workers of training jobs (TFJobs), release sufficient GPU resources for inference pods. We leave at least one worker (chief worker) for every TFJobs in our implementation, so that TFJobs can proceed training process continuously without requiring checkpoint. Finally, the Kubernetes API scheduler would allocate GPU resources for the new inference pods, which makes platform reach inference jobs' Service Level Agreement (SLA).

In our implementation, we label the inference pods first, making Kubernetes Operator capable of identifying inference jobs. Then, by Kubernetes native event-driven mechanism, we registered OnAdd, OnUpdate events of inference pods. Whenever Kubernetes HPA decides to scale up inference pods, the training job scheduler would be notified by the Kubernetes control manager, check whether the current residual resource is available for the new pod, and decide whether to release the resources for inference jobs by scaling down training workloads. Also, training job scheduler would periodically check the residual resources. After Kubernetes HPA scale down inference pods and release the GPU resources, training job scheduler would scale up training jobs once detect residual resources.

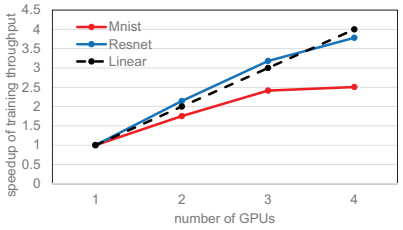

Figure 3: The speedup of model training throughput of using multi-GPUs.

4 Experiment Setup

We evaluate our implementation by conducting the experiments on AWS cloud platform using a Kubernetes cluster consisting of 2 nodes (p3.16xlarge instance type). Each node is equipped with a 64-cores CPU (Intel Xeon E5-2686 v4), 488GB of RAM, and 8 Nvidia Tesla V100 GPUs with 128GB of device memory. In order to conduct a comprehensive evaluation of our implementation, we design workloads that include the computing jobs for modeling, training, and inference. The training and inference jobs are based on the popular DL framework - Tensorflow. The modeling jobs are based on the Jupyter Notebook, which is a primary tool for developers to build and test their models.

For the training jobs, we employ two different common-seen image classification models (Mnist, ResNet-50) and divide all of the Tensorflow jobs (TFJob) into three groups. Every TFJob would consist of one Parameter Server (PS) and several workers, where workers can be added or removed in response to the state of the residual resource in our implementation. Each worker requests one GPU, and a TFJob runs for a fixed training iterations. As shown in Figure 3, both models can have higher training throughput and shorter training time when using more GPUs. The speedup of ResNet-50 is close to linear. The speedup of Mnist doesn't increase much with more than 3 GPUs because its model is too small to have enough computations for parallel processing. In the experiments, we set the maximum workers to 4 for both models.

For the inference jobs, we use a Resnet-50 model as our inference service for all requests. Inference job runs on TF-Serving applications, which computes the forward propagation upon each arrival client requests. Hence, its GPU usage is also approximately proportional to the number of client requests. In our experiment, One TF-Serving inference pod is configured to consume only one GPU. As a result, the number of inference pods would dynamically increase or decrease in response to the number of active clients. For the purpose of evaluating the response time of inference

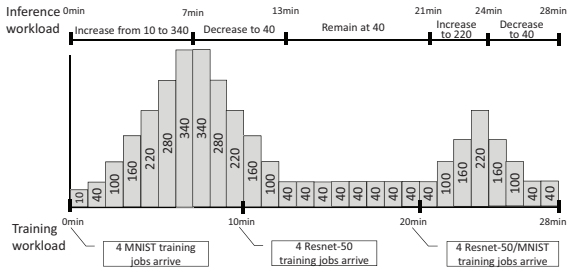

Figure 4: Testing workload for system evaluations. The workload of interface jobs are time-varied by adjusting the number of active users. The workload of training jobs are submitted in 4 groups and 4 jobs per group at time 0, 10, 20. The modeling jobs are consisted of 4 notebook instances persistently running throughout the experiments.

Table 2: Compared system configurations. The setting with a resource management technique is marked by "V".

	Training scheduling	Inference auto-scaling	GPU Sharing
DynamoML	V	V	V
K8S+Sharing			V
K8S+Scaling		V	
Native K8S			

job under different workloads, a web client would keep sending requests with a varied inter-arrival time.

Finally, for the modeling jobs, we sporadically issue model evaluation requests to the Jupyter Notebook instances, and the average GPU usage of a modeling job never exceeds 25%.

To evaluate the benefits of our resource management techniques in a runtime system. We construct a testing work as shown in Figure 4. It contains a total of 4 modeling (notebook) jobs, 1 inference job, and 16 training jobs. The inference jobs and modeling jobs are persistently running in the system, but their user workload changes over time. As mentioned above, the modeling has a sporadic random generated workload with less than 25% GPU usage. The workload of the inference job controlled by adjusting the number of concurrent active web clients in each time interval (per minute) which can be seen from the bars at each minute in Figure 4. Finally, the 16 training jobs are submitted to the system every 10minutes in 4 groups with 4 jobs per group. The first group arrives at 0^{th} minute with 4 Mnist training jobs. The second group arrives at 10^{th} minute with 4 Resnet-50 training jobs. The last group arrives at 20^{th} minute with 2 training jobs for each of the two models.

The goal of our evaluation is to compare the system and application performance running the above test work under different resource management configuration settings. The names and resource management techniques of each setting are summarized in Table 2. The initial number of workers for a training job is 2, and the initial number of server instance for a inference job is 1. Both training and inference jobs can be scaled upto to 12 instances (training workers or inference servers) when auto-scaling techniques are applied (i.e., the total number of GPUs in our testbed is 16.). By default, all types of jobs request one GPU per container instances (i.e., pods). Only when GPU sharing technique is applied, a modeling job can allocate 0.25 GPU.

5 Experiment Results

5.1 System Performance Comparison

Figure 5 plots the GPU resource allocation results over each time interval under different system settings. Due to space limits, we show the results of DynamoML and K8S+Scaling to illustrate the key benefit of our approach. As recall, DynamoML supports all three proposed techniques: inference scaling, training scheduling, and GPU sharing. K8S+Scaling only supports inference scaling. K8S+Scaling also represents the common use case when people are only able to use the native Kubernetes installation with the HPA auto-scaling package to run ML workloads.

According to the workload variation of inference job, the allocation result can be discussed in the following three time frames from the DynamoML timeline.

0min∼7min: The inference workload keeps increasing during this time frame. Our auto-scaling operator detected the increased request response time, and started to launch more inference servers. Hence, the number of GPUs allocated to inference increases from 1 to 12. Noted, the inference job can occupy as many as 12 GPUs because the training jobs are preempted and forced to release their GPUs for inference jobs. Therefore, some the training job only allocated 1 GPU at times. On the other hand, when there is residual capacity or un-used GPUs, they can also be dynamically allocated to the training jobs for reducing training time. Therefore, the GPU allocation during this time frame is almost always fully utilized.

8min∼20min: As the inference workload decreases, we can observe DynamoML quickly allocate the available GPUs free from the inference job to the training jobs. As a result, a training job can use up to 4 GPUs at a time, and the training time is greatly reduced. Because the training job finishes too early, only 4GPUs needs to be used for handling the inference requests between time 18min∼20min which can lead to an additional benefit of energy and cost saving for system administrators or service providers.

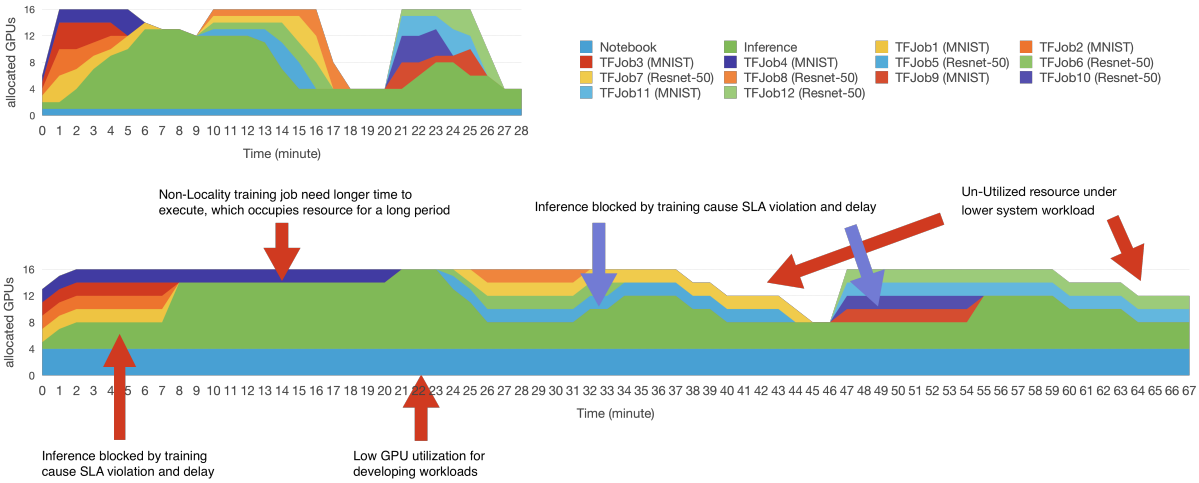

Figure 5: The overall resource allocation for DynamoML (top) and K8S+Scaling (bottom).

21min∼28min: The last group of training job arrives at 20min, and the inference workload also starts to increase at 21min. So similar to the first time frame, both training jobs and inference jobs can get more resources at runtime, but the inference jobs have higher priority than the training jobs. Also because of the lower inference workload in this time frame comparing to the first time frame, training jobs received more GPUs and complete all the training jobs before 27min.

In comparison, we can observe several problems from the K8S+Scaling setting. (1) It fixes the allocation of training job to 2 GPUs, and each modeling job (Notebook instance) occupies 1 GPU. Hence, even though the inference job can be scaled to obtain more GPUs, it can only use the residual capacity from training and modeling jobs. Therefore, between 0min to 7min, the inference job only receives 4 GPUs while it would receive 11 GPUs by DynamoML. (2) The Kubernetes scheduler didn't pack the workers of a training on the same node which results in much longer training time. In particularly, the communication overhead has a greater impact on the small size models, like MNIST, because their communication time often takes higher ratio of the overall execution time. Therefore, compared to the results of DynamoML, TFJob4, TFJob7 and TFJob11 all took much longer time to finish under the K8S+Scaling setting. (3)Because the resource of training jobs is fixed, they can take advantage of the residual capacity in the system when system workload is light. For instance, there are GPUs available between 37min to 46min, and 59min to 67min, but they cannot be allocated to training jobs and cause unwanted resource waste.

In sum, we can observe the important of dynamic resource management when running complex and diverse workloads on a shared resource pool. With our techniques, the overall workload execution time is significantly reduced from 67mins to 27mins, an improvement of almost 60%, and there are still rooms for us to free-up some idle resources for energy or cost saving. More importantly, DynamoML can improve resource utilization and training performance while guaranteeing the SLA requirement of inference jobs. In the next two subsections, we further analyze the performance of training jobs and the SLA violations of inference jobs to analyze the reasons of our improvement.

5.2 Training Time Analysis

This subsection analyzes the impact of our resource management techniques on the training jobs from running test workload shown in Section 4. Because training job has lower execution priority and can be queued in the submission queue when not enough resource available, the total execution time of a training can be divided into two parts: the training time and the waiting time. The training time is the actual running time for training, and the waiting time is time of a job waiting in the scheduling queue. Therefore, we compare the improvements of these two time measurements, and the total execution time in Figure 6. Interestingly, we found that K8S+Scaling produces the worst results across all the measurements, even worse than the native K8S. This is because K8S+Scaling only optimizes for the inference jobs not for the training jobs. Hence, in order to satisfy the SLA requirements, it allow inference jobs to allocate more resources, and sacrifices training jobs. As a result, training jobs have higher probability to be

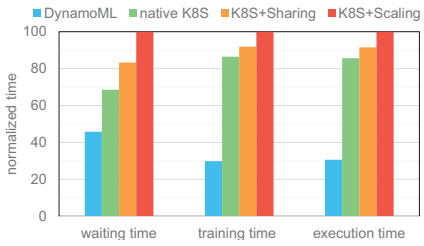

Figure 6: Training time comparison.

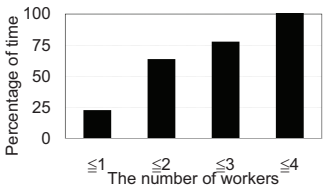

Figure 7: The distribution of the number of worker per TFJob under DynamoML.

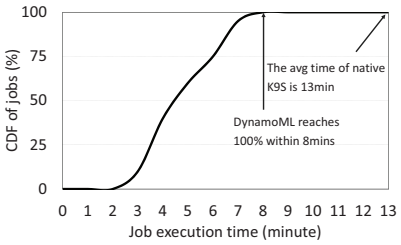

Figure 8: The distribution of the training job execution time under DynamoML.

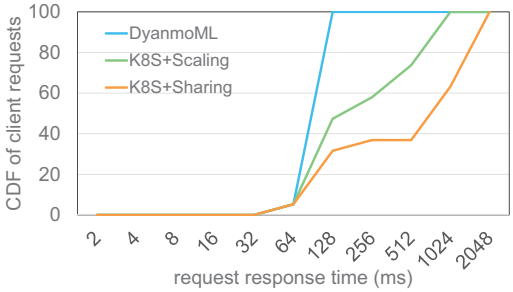

Figure 9: The comparison of the distribution of the response time from inference jobs.

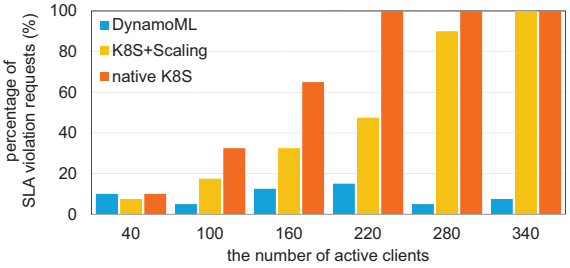

Figure 10: SLA Violation of inference jobs under different number of active users

blocked or running with few GPUs. K8S+Sharing is better than native Kubernetes, because it reduces the amount of GPUs used by the modeling jobs, so training jobs can gain better performance. However, the improvement of GPU sharing for training job is limited, because the communication overhead is the main performance bottleneck for training jobs as we saw from the cases like TFjob4 in Figure 5. In comparison, DynamoML combines all our techniques to significantly reduce the waiting time by 55%, the training time by 70%, and the total execution time by 70%. Finally, to prove our training job scheduler did dynamically add or remove workers to training jobs according to the system loading, Figure 7 shows the time distribution of a training job with a given number of workers when using DynamoML. As seen, in average, about 22% of the job execution time uses only 1 GPU, 63% of the time uses 2 or less GPUs. Only the reminding 37% of time using 3 or 4 GPUs. But according to the execution time distribution shown in Figure 8, DynamoML still significantly reduce the overall training time. With DynamoML, more 60% of the training jobs finishes within 5mins, and the longest execution time is 8mins. In contrast, the average execution time for native K8S is 13mins.

5.3 Inference Performance Analysis

Lastly, we analyze the SAL violation of inference job from the running test workload shown in Section 4. Figure 9 shows the overall client response time distribution of the TF-serving inference job under different system settings. K8S+Sharing has the worst results, where some of the requests have response time over 1024ms. This is because it doesn't supports auto-scaling on the inference service. DynamoML performs the best with no requests with response time over 128ms, because it can preempt training jobs when necessary. One the other hand, although K8S+Scaling also supports auto-scaling on inference inference, but it cannot preempt training jobs. As a result, the amount of resources for inference jobs can be bounded by the training jobs. Therefore, the response time of K8S+Scaling is mostly between 128ms∼512ms. Figure 10 further breaks down the SLA violation probability under different inference workloads which is controlled by the number of active clients. As expected, the violation probability increases under highly workload. Only DynamoML can be resilient to the workload because its ability to obtain enough resources to satisfy the SLA requirements.

6 Related Work

In recent years, both research and industry have made great efforts to improve the performance of deep learning jobs in a GPU cluster by utilizing domain-specific knowledge. However, all these works target on distributed training jobs alone. While our work address the ML pipeline workload including model-

ing and inference jobs as well. Also, most of the proposed techniques requires modifications to the deep learning frameworks, while our work can be transparent and general to DL applications.

Gandiva (Xiao et al., 2018) is a scheduling framework developed by Microsoft. It supports many management techniques together to maximize system throughput for training jobs. It provides GPU sharing among jobs in both temporal and spatial domains. In temporal domain, jobs run on GPU in an interleaved manner through suspend-resume mechanism. In spatial domain, jobs simply run simultaneously on a GPU at the same time, but jobs can be migrated to another GPU if performance degradation is detected. Gandiva also applies scaling mechanism to jobs that self-declare to have good scalability. In order to minimize the overhead of their managing overhead like migration, suspend-resume, and scaling, Gandiva has to modify the deep learning frameworks, like Tensorflow and Pytorch.

The rest of researches are mainly focus on a more specific resource management technique. Some of the work focus on scaling policy of distributed training jobs. For instance, Optimus (Peng et al., 2018) proposes a GPU resource scheduler to decide the proper resource amount and resource allocation according to a performance model of distributed training jobs. Similar to Optimus, DL2 (Peng et al., 2021) is a DL driven scheduler that aims to decide proper resource allocation to a distributed training job. But the performance model proposed by DL2 is based on deep reinforcement learning.

Other works focus on the job placement problems for minimizing the communication time of distributed training jobs. For instance, (Amaral et al., 2017) proposed a topology aware scheduling to decide the mapping between worker tasks and GPU slots based on the Hierarchical Static Mapping Dual Recursive Bipartitioning algorithm. Rather than optimizing the performance of a single training job, Tiresias (Gu et al., 2019) is a scheduler based on the Gittins index policy to minimize the average job completion time of the whole system.

Finally, there is growing interest to explore GPU sharing technique for DL jobs. Spatial GPU sharing can suffer from unpredictable performance interference and resource contention. So, temporal GPU sharing is more commonly adapted in practice. But temporal sharing can be limited by the GPU memory size, and context switch overhead. Salus (Yu and Chowdhury, 2019) takes advantage of the highly predictable and largely temporal usage memory pattern to provide a fine-grained sharing mechanism by switching jobs at the lowest memory usage point. Antman (Xiao et al., 2020) further modifies the execution and scheduling engine of deep learn frameworks to support switching at the unit of operators (GPU kernels).

7 Conclusions

Deep learning workflow has become one of the primary workloads in data centers and GPU clusters. In this paper, we aim to optimize the application performance and system utilization through a set of runtime dynamic resource management techniques. We use GPU sharing to increase the resource utilization of non-GPU bounded modeling jobs, use performance-driven auto-scaling to guarantee the SLA requirement of inference jobs, and use workload-aware scheduling and preemption to improve training job efficiency and avoid idle GPUs. While all these techniques have been discussed and used for different kinds of computing workload, we are one of the few work that really integrate and apply them together specifically for the ML pipeline workflow. Our system are built as extended operators on Kubernetes, and transparent to applications. Hence, our solution can be easily applied to general GPU clusters and DL workload. While this work mainly aims to demonstrate the benefit and importance of dynamic resource management to ML workload, we plan to evaluate our system with more complex and real ML pipeline workload to further study and design more sophisticate algorithms for each of optimization techniques in the future.

REFERENCES

Amaral, M., Polo, J., Carrera, D., Seelam, S., and Steinder, M. (2017). Topology-aware gpu scheduling for learning workloads in cloud environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12.

Amodei, D. and Hernandez, D. (2018). Ai and compute. https://openai.com/blog/ai-and-compute/.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2018). BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. (2011). Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, page 323–336.

Gu, J., Chowdhury, M., Shin, K. G., Zhu, Y., Jeon,

M., Qian, J., Liu, H., and Guo, C. (2019). Tiresias: A GPU cluster manager for distributed deep learning. In *NSDI*, pages 485–500.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

He, X., Pan, J., Jin, O., Xu, T., Liu, B., Xu, T., Shi, Y., Atallah, A., Herbrich, R., Bowers, S., and Candela, J. Q. n. (2014). Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, page 1–9.

Jalaparti, V., Bodik, P., Menache, I., Rao, S., Makarychev, K., and Caesar, M. (2015). Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, page 407–420.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. (2019). ALBERT: A lite BERT for self-supervised learning of language representations. *CoRR*, abs/1909.11942.

Levine, S., Pastor, P., Krizhevsky, A., and Quillen, D. (2016). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *CoRR*, abs/1603.02199.

Lin, C.-Y., Yeh, T.-A., and Chou, J. (2019). DRAGON: A dynamic scheduling and scaling controller for managing distributed deep learning jobs in kubernetes cluster. In *International Conference on Cloud Computing and Services Science (CLOSER)*, pages 569–577.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

Peng, Y., Bao, Y., Chen, Y., Wu, C., and Guo, C. (2018). Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*, pages 1–14.

Peng, Y., Bao, Y., Chen, Y., Wu, C., Meng, C., and Lin, W. (2021). DL2: A deep learning-driven scheduler for deep learning clusters. *IEEE Transactions on Parallel Distributed Systems*, 32(08):1947–1960.

Redmon, J., Divvala, S. K., Girshick, R. B., and Farhadi, A. (2015). You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640.

Tannenbaum, T., Wright, D., Miller, K., and Livny, M. (2001). Condor – a distributed job scheduler.

Tian, Y., Pei, K., Jana, S., and Ray, B. (2017). Deeptest: Automated testing of deep-neural-network-driven autonomous cars. *CoRR*, abs/1708.08559.

Tumanov, A., Zhu, T., Park, J. W., Kozuch, M. A., Harchol-Balter, M., and Ganger, G. R. (2016). Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*, pages 1–16.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008. Curran Associates, Inc.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. (2013). Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of Symposium on Cloud Computing*.

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-scale cluster management at google with borg. In *EuroSys*, pages 1–17.

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., Yang, F., and Zhou, L. (2018). Gandiva: Introspective cluster scheduling for deep learning. In *OSDI*, pages 595–610.

Xiao, W., Ren, S., Li, Y., Zhang, Y., Hou, P., Li, Z., Feng, Y., Lin, W., and Jia, Y. (2020). Antman: Dynamic scaling on GPU clusters for deep learning. In *OSDI*, pages 533–548.

Xu, D., Anguelov, D., and Jain, A. (2017). Pointfusion: Deep sensor fusion for 3d bounding box estimation. *CoRR*, abs/1711.10871.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., and Le, Q. V. (2019). Xlnet: Generalized autoregressive pretraining for language understanding. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 32, pages 5753–5763. Curran Associates, Inc.

Yeh, T.-A., Chen, H.-H., and Chou, J. (2020). Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, page 173–184.

Yu, P. and Chowdhury, M. (2019). Salus: Fine-grained GPU sharing primitives for deep learning applications. *CoRR*, abs/1902.04610.

Zoph, B., Cubuk, E. D., Ghiasi, G., Lin, T., Shlens, J., and Le, Q. V. (2019). Learning data augmentation strategies for object detection. *CoRR*, abs/1906.11172.