Job Scheduling for Large-Scale Machine Learning Clusters

Haoyu Wang, Zetian Liu, Haiying Shen
Department of Computer Science, University of Virginia
{hw8c,zl4dc,hs6ms}@virginia.edu

ABSTRACT

With the rapid proliferation of Machine Learning (ML) and Deep learning (DL) applications running on modern platforms, it is crucial to satisfy application performance requirements such as meeting deadline and ensuring accuracy. To this end, researchers have proposed several job schedulers for ML clusters. However, none of the previously proposed schedulers consider ML model parallelism, though it has been proposed as an approach to increase the efficiency of running large-scale ML and DL jobs. Thus, in this paper, we propose an ML job Feature based job Scheduling system (MLFS) for ML clusters running both data parallelism and model parallelism ML jobs. MLFS first uses a heuristic scheduling method that considers an ML job's spatial and temporal features to determine task priority for job queue ordering in order to improve job completion time (JCT) and accuracy performance. It uses the data from the heuristic scheduling method for training a deep reinforcement learning (RL) model. After the RL model is well trained, it then switches to the RL method to automatically make decisions on job scheduling. Furthermore, MLFS has a system load control method that selects tasks from overloaded servers to move to underloaded servers based on task priority, and also intelligently removes the tasks that generate little or no improvement on the desired accuracy performance when the system is overloaded to improve JCT and accuracy by job deadline. Real experiments and large-scale simulation based on real trace show that MLFS reduces JCT by up to 53% and makespan by up to 52%, and improves accuracy by up to 64% when compared with existing ML job schedulers. We also open sourced our code.

CCS CONCEPTS

• **Software and its engineering** → **Cloud computing**;

KEYWORDS

job scheduling, resource management, machine learning

ACM Reference Format:
Haoyu Wang, Zetian Liu, Haiying Shen. 2020. Job Scheduling for Large-Scale Machine Learning Clusters. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3386367.3432588

CoNEXT '20, December 1–4, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7948-9/20/12…$15.00
https://doi.org/10.1145/3386367.3432588

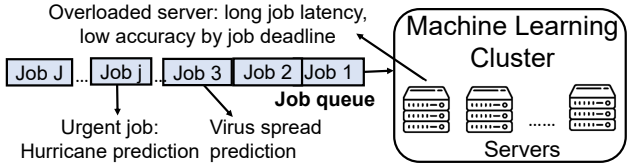


Figure 1: ML job cluster.

1 INTRODUCTION

The rapid proliferation and development of Machine Learning (ML) and Deep Learning (DL) techniques have paved the way for a future where intelligent applications can assist people with personal tasks such as living, working, and learning, as well as broader community goals such as more efficient transportation, effective healthcare, and other activities. For example, scientists use ML techniques to predict the spread of a virus (e.g., coronavirus in Wuhan China in 2020) among the population, and the path of a hurricane (e.g., Hurricane Dorian in 2019) for in-time precaution in affected areas. Many companies (e.g., Amazon and Google) and academic organizations now provide platforms for users to train their ML models. As shown in Figure 1, ML jobs are submitted to an ML cluster and put into a job queue where they wait to be assigned to servers. Jobs always have performance requirements (e.g., deadline and accuracy). For example, an ML job for predicting a hurricane path must be completed by a certain time before the hurricane landfall with a high prediction accuracy to maximally avoid damage and casualties. Therefore, it is crucial to ensure that all running and waiting ML jobs in an ML cluster complete in time without degrading their accuracy performance.

In recent years, the ML and DL models have been growing deeper and larger, which dramatically increases the job completion time (JCT). To meet the high-scalability requirement of increasingly large-scale ML or DL applications, model parallelism has been proposed [18, 25, 27, 30], in which an ML model is partitioned, and multiple partitions run in parallel. As the scale of ML and DL models rapidly grows [22], a job scheduler for model parallelism is increasingly necessary to help reduce the JCTs of jobs while guaranteeing a high degree of accuracy. However, to the best of our knowledge, none of the previously proposed schedulers can handle both model and data parallelism in spite of the previous research on job scheduling in ML clusters [11, 21, 29, 32, 39, 42, 43, 45, 52, 55, 56, 58].

Therefore, in this paper, we propose a Machine Learning job Feature based job Scheduling system (MLFS) for both data parallelism and model parallelism ML jobs in an ML cluster. In addition, MLFS is also applicable for data parallelism only ML jobs and model parallelism only ML jobs. MLFS should meet the following requirements. First, it should reduce the bandwidth cost, which can be the bottleneck of distributed ML jobs [33, 57] when enhancing JCT, where frequent data transmission between tasks is needed. For example,

the communication overhead between GPUs is 970MB-3168MB per mini-batch for data parallelism jobs and 784MB-1037MB per mini-batch for model parallelism jobs [12]. Second, it should concurrently improve JCT and accuracy since both are important for ML jobs. For example, hurricane path prediction jobs are time critical and also require high accuracy to avoid damage and casualties. Third, it should prioritize jobs by considering their urgency and accuracy requirements. For example, hurricane path prediction is time-critical and requires high accuracy, but annual people movement prediction is time tolerant and does not require very high accuracy. Fourth, it can still provide job deadline guarantee or low JCT when the system is overloaded.

In summary, the goal of MLFS is to *minimize the average JCT, maximize the average job accuracy, maximize the number of jobs whose deadline and accuracy requirements are satisfied, and minimize the bandwidth cost.* MLFS is novel in that it intelligently takes advantage of the spatial/temporal features of ML jobs. First, different tasks (for different model partitions) run separately, and the tasks have different impacts on the final job latency and accuracy (i.e., spatial features). For example, in the ML model partition graph (or task dependency graph), if a task has more dependent tasks, or its dependent tasks are in layers closer to the task, it should run earlier to improve job latency and accuracy by the job deadline. In addition, the model partition size (measured by the number of ML model parameters) influences the final accuracy result – a larger size generates a higher impact and vice versa. Second, an ML job usually runs many iterations, and earlier iterations have higher impact on the accuracy than later iterations [58] (i.e., temporal features). Therefore, the tasks in earlier iterations should have a higher priority to run and vice versa. Accordingly, MLFS first uses a heuristic scheduling method that considers these features to determine task priority for job queue ordering in order to improve the JCT and accuracy performance. To relieve extra load from an overloaded server, it also considers the task priority in selecting tasks to move out from the server and puts them in the queue to be rescheduled. It uses the data from the heuristic scheduling method for training a deep reinforcement learning (RL) model. After the RL model is well trained, it then switches to the RL method to automatically make decisions on job scheduling. Furthermore, when the system is overloaded, MLFS has a system load control method that selects tasks from overloaded servers to move to underloaded servers based on task priority and also intelligently terminates the tasks that generate little on the required accuracy performance, which helps improve both JCT and accuracy by the job deadline.

MLFS consists of the following components:

(1) **ML feature based heuristic task scheduling (MLF-H).** MLF-H determines the task priority based on the spatial/temporal ML job features and computation features (e.g., deadline, waiting time, remaining job time) to improve both JCT and accuracy. Tasks that contribute more to the accuracy and JCT are given higher priorities. The priority of a task is used for queuing tasks and allocating tasks when there are servers with available resources. MLF-H also handles overloaded servers by considering task priority and other factors in selecting tasks to migrate out of overloaded servers, and reschedules these tasks along with the waiting tasks in the queue. MLF-H further tries to fully utilize resources and reduce bandwidth cost when selecting migration tasks from an overloaded server and when selecting a host server to allocate a task.

(2) **ML feature based RL task scheduling (MLF-RL).** MLFS initially runs MLF-H for a certain time period and uses the data to train a deep RL model, and it then switches to MLF-RL when the model is well trained. Given both running and waiting tasks as well as nodes, based on their status, MLF-RL selects tasks to move out of overloaded nodes and determines the destination node or the queue for the selected tasks and the waiting tasks in the queue to achieve the aforementioned goal.

(3) **ML feature based system load control (MLF-C).** When the system is overloaded, jobs will experience high JCT and low accuracy by the job deadline. MLF-C can stop running or generating tasks once the desired accuracy is reached (based on users' choices) to relieve system workload in order to improve JCT and accuracy by the job deadline. MLF-C also adopts an optimal ML iteration stopping method that finds the iteration to stop training in order to achieve the maximum accuracy while minimizing the number of iterations.

We conducted real experiments using Pytorch [4] on AWS [40] and large-scale simulation based on real workload trace [3]. Extensive experimental results show the superior performance of MLFS compared to state-of-the-art methods in [21, 39, 53, 55, 58] and the effectiveness of each of its components. We open-sourced our code in Github [5].

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 presents our goal and the details of MLFS. Section 4 presents performance evaluation. Section 5 presents the summary of the experiment results and the limitations of our methods. Section 6 presents our conclusions and our future work.

2 RELATED WORK

A significant amount of research effort has been devoted to job scheduling in clusters and cloud datacenters (e.g., [9, 13–16, 19, 34, 37]). In this paper, we focus on job scheduling for ML clusters [21, 24, 39, 49, 53, 55, 58]. TensorFlow [8] uses the Borg resource manager [53] that aims to achieve fairness of resource allocation among different jobs. SLAQ [58] aims to maximize the overall job accuracy. For the available CPU resource, SLAQ predicts the loss reduction and runtime if different numbers of CPU cores are assigned to each of the running jobs, and then chooses the job with the maximum loss reduction per unit runtime to adjust the number of CPU cores. Tiresias [21] is proposed to schedule DL jobs in a GPU cluster to reduce JCT. It determines job priority (for queuing and preemption when there are no available GPUs) using two principles: 1) for jobs without prior knowledge of its task running time, the least-attended service principle gives higher priorities to the jobs that received less service time; and 2) for jobs with known task running time distribution on GPUs, the priority is determined by how likely the job can complete within the next service epoch if it receives the available GPUs. Gandiva [55] uses first-in-first-out (FIFO) queuing. Also, it defines the jobs with the same number of GPU requirements as affinity jobs and tries to put the affinity jobs to the same machine to more fully utilize resources. Meanwhile, to relieve the extra load of an overloaded GPU (i.e., GPU utilization is higher than a threshold), Gandiva moves the job with

the lowest GPU utilization to the GPU with the lowest utilization to avoid resource fragmentation and achieve high resource efficiency. Based on Gandiva, Gandiva-fair [11] was proposed, which is a distributed fair share scheduler that balances conflicting goals between efficiency and fairness of resource allocation among users. Optimus [42] uses online fitting to predict an ML job's convergence performance during training and estimates the training speed as a function of assigned resources on each job. Then, it tends to assign more resources to the job which will be finished sooner in order to reduce the average JCT while providing accuracy guarantee. HyperDrive [45] tends to give high priorities for resource allocation to those jobs that have higher possibilities to achieve better accuracy performance (based on the ML job model accuracy prediction) while reducing JCT. HyperSched [32] aims to produce a trained model with higher accuracy before the pre-set deadline under a certain resource constraint. This method pauses jobs that do not increase accuracy significantly and tends to assign more resources to the job with more accuracy improvement before its deadline. AlloX [29] transforms the job scheduling problem into a min-cost bipartite matching problem in order to reduce the average JCT. TetriSched [52] aims to improve the cluster utilization and also the number of jobs that are completed before their pre-set deadlines. It formulates the job scheduling problem into an optimization problem and uses integer linear programming to solve it. Yabuuchi *et al.* [56] proposed an algorithm that first identifies whether a job is trial-and-error or best-effort and then tends to give more resources to best-effort jobs in order to reduce the JCT. The trial-and-error jobs aim to find the best hyperparameter configurations, and the best-effort jobs try to achieve the best accuracy improvement with fixed hyperparameter configuration. DL2 [43] is an RL-based job scheduling algorithm which uses job completion time as a reward directly. Dorm [49] is a cluster management system that uses a container-based virtualization technique to partition a cluster and then runs one application per partition to increase resource utilization and speed up ML jobs. Narayanan *et al.* [41] proposed *PipeDream*, a method that uses pipeline parallelism to enable faster DNN training by combining intra-batch parallelism with inter-batch parallelization within one job. Grandl *et al.* [20] proposed Graphene, a job scheduler for general jobs with a task dependency DAG (directed acyclic graph). Within one job, Graphene tends to first assign the available resources to the "troublesome" tasks (the tasks have more dependent tasks and tough-to-pack resource demands (e.g., high resource demand of one job) and then assign the remaining resources to the remaining tasks without dependency violation. For a set of jobs, Graphene determines the order of multiple jobs based on a weighted scores calculated based on multiple job scheduling objectives including average job completion time, cluster throughput and fairness. Mirhoseini *et al.* [39] applied RL in job scheduling in a GPU cluster to minimize the average JCT. The scheduler scans all tasks and then maps the tasks to the appropriate GPUs. In the past few years, RL has been used for job scheduling for general jobs in clusters to improve the average JCT [35, 37].

Compared with the above job schedulers, MLFS is novel in that it leverages the ML job spatical/temporal features in job scheduling to improve their JCT and accuracy.

3 SYSTEM DESIGN

3.1 Assumptions

MLFS can be used for all types of ML workloads, and it significantly benefits ML jobs running in the model parallelism manner. Based on the prior research, we make several assumptions. First, we assume that the accuracy of a job can be predicted. We adopt the previously proposed method in [17] for the accuracy estimation that achieves around 90% accuracy. We will explain this method in Section 3.5. The accuracy at a certain iteration is predicted based on the number of iterations executed and the accuracy change for each executed epoch. Therefore, a user does not need to enter the details of his/her job for the accuracy prediction in our system.

We also assume that the total job running time can be predicted accurately. We use the approach in [42] for the running time prediction. It achieves 89% prediction accuracy for the jobs that ran previously and 70% prediction accuracy for the jobs that didn't run previously. We assume that the task dependency graph of each job is pre-known [23]. Our methods can be used for both GPU nodes and CPU nodes, and we use GPUs as an example unless otherwise specified. We assume that each user specifies the job deadline and accuracy requirements for his/her jobs. For a job without such specifications, the system will assign the maximum deadline as the job deadline and the minimum accuracy as the accuracy requirement. Further, we assume that RL is trained based on all the previously running models for all considered resource types.

3.2 Problem Formulation

Currently, TensorFlow uses data parallelism, which divides a training dataset to multiple mini-batches, and each mini-batch (task) runs in a worker (i.e., a computing slot). We use the complex scenario with both data parallelism and model parallelism to present our methods. Our MLFS also can be applied to the data parallelism only, model parallelism only and none parallelism scenarios. In the scenario of both data parallelism and model parallelism (as shown in Figure 2), a task (running in a worker) computes one model partition for one mini-batch. The tasks form a task dependency graph based on the data flow between the tasks. The parameter server communication structure and the all-reduce communication structure are used to accumulate the learned parameters in training. In the parameter server communication structure, each task sends its training results to the parameter server, which accumulates the results and updates the ML model. To apply the data and model parallelism to the parameter server communication structure, an ML network is divided into different partitions and each partition runs in a worker and handles one mini-batch. The workers transmit data based on the ML partition dependency, and the final workers in the dependency graph (i.e., model partition graph) send the training results to the parameter server. In the all-reduce communication structure [46], one parameter server containing the whole model and one worker are combined together as one reducer, which is the unit execution processor. For the model parameter update in each iteration, each reducer communicates to other reducers to update model parameters using a communication topology (e.g., ring all-reduce [7] and 2D-Torus [38]). To apply data and model parallelism to the all-reduce communication structure, we can directly use the existing communication topology for the parameter

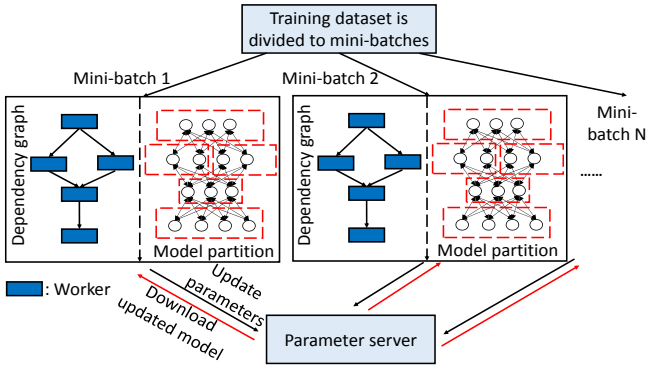


Figure 2: ML platform with data parallelism and model parallelism.

transmission between workers to update their models. Therefore, the task dependency graph in data and model parallelism can be for both parameter server and all-reduce communication structure.

Assume that $\mathcal{J}$ represents the job set, $\mathcal{T}$ represents the task set, and $\mathcal{N}$ represents the node set and the queue. We use (k, n) to represent that task k is allocated to node n or the queue. Our goal is to find a task allocation plan $A = \{(k, n)|k \in \mathcal{T}, n \in \mathcal{N}\}$ that minimizes the average JCT, maximizes the average ML model accuracy, maximizes the number of jobs whose deadline and accuracy requirements are satisfied, and minimizes the bandwidth cost between nodes. For job J, we use d_J to denote its JCT, d_J^r its deadline, a_J its final accuracy, a_J^r its accuracy requirement. Then, the goal can be represented by the formula below:

$$\begin{cases} g_1(A) = & 1/\frac{\sum_{J\in\mathcal{J}} d_J}{|\mathcal{J}|} \\ g_2(A) = & \sum_{J\in\mathcal{J}} \mathbb{1}(d_J^r \geq d_J) \\ g_3(A) = & 1/\sum_{n_i,n_j\in\mathcal{N}} B_{n_i,n_j} \\ g_4(A) = & \sum_{J\in\mathcal{J}} \mathbb{1}(a_J \geq a_J^r) \\ g_5(A) = & \frac{\sum_{J\in\mathcal{J}} a_J}{|J|} \end{cases}$$

$$max\ (g_1(A), g_2(A), g_3(A), g_4(A), g_5(A)), \quad (1)$$

in which $\mathbb{1}$ is a binary indicator function, and B_{n_i,n_j} is overall size of the data transmitted between n_i and n_j. The goal is to maximize the value of each objective function $g_i(A)$ $(i = 1, 2, ..., 5)$ simultaneously. Note that this problem formulation covers both all-reduce communication structure and parameter server communication structure. This is a multi-objective optimization problem. We could use the adaptive epsilon constraint algorithm [28] to solve the above multi-objective optimization problem. However, due to its high computation overhead, we instead propose a heuristic task scheduling method to obtain the solution of this optimal problem and an RL-based task scheduling method in the following sections.

3.3 ML Feature Based Heuristic Task Scheduling (MLF-H)

3.3.1 Priority Determination. We use $P_{k,J}$ to denote the priority of task k of job J which is in its I^{th} iteration. We use I_{max} to denote the specified maximum number of iterations of job J. We first introduce how we consider the ML spatial and temporal features to determine the priority (denoted by $P_{k,J}^{ML}$), and then introduce how we consider the traditional computation features to determine the priority (denoted by $P_{k,J}^{C}$). Finally, we combine the two priority values to calculate $P_{k,J}$.

First, the jobs running in a cluster have different urgency levels, as explained in Section 1. The system sets m levels of job urgent coefficients $L^J \in [0, m]$; a higher urgent coefficient means that the job is more urgent and vice versa. The urgent coefficients of jobs in a cluster can be pre-determined by the system administrator based on the urgency of the applications or specified by the users. A higher urgent coefficient specified by a user leads to higher monetary cost to the user. Giving higher priorities to jobs with higher urgency levels can help the jobs to meet their deadline requirement.

Second, for an ML job, based on the ML temporal features, the earlier iterations are more important than later iterations on improving accuracy because of the diminishing loss reduction returns [58]. Therefore, tasks that will gain higher accuracy improvement in the next iteration should have a higher priority to be scheduled in order to improve the average accuracy of jobs in the system. We use the inverse proportional function $\frac{1}{I}$ $(I \geq 1)$ to represent the importance of the current iteration of the job. A larger ratio $\frac{1}{I}$ means an earlier stage of the job running, so the I^{th} iteration contributes more on accuracy improvement of the ML job. We use δl_{I-1} to denote the loss reduction of the most recent finished iteration $I-1$. Then, $\sum_{j=1}^{I-1} \delta l_j$ denotes the overall loss reduction achieved by all the completed iterations, and $\frac{\delta l_{I-1}}{\sum_{j=1}^{I-1} \delta l_j}$ represents the normalized loss reduction of the most recent completed iteration. A higher value of $\frac{\delta l_{I-1}}{\sum_{j=1}^{I-1} \delta l_j}$ means that the I^{th} iteration contributes more on the loss reduction. Note that these two formulas represent the trends of general ML jobs and can be replaced by other appropriate formulas specific to certain ML jobs.

Third, considering the ML spatial features in model parallelism, a larger model partition usually plays a more important role in calculating the model parameters. We measure the model partition size by the number of ML model parameters in this model partition. We use $S_k^J = S_k/S^J$ to denote the normalized size of the model partition of task k, where S_k is the size of the model partition and S^J is the entire ML model. Giving a higher priority to a larger model partition can contribute more in increasing the model accuracy.

Combining all of these ML features, for task k that has no dependent tasks, its priority is calculated by:

$$P'^{ML}_{k,J} = L^J \cdot \frac{1}{I} \cdot \frac{\delta l_{I-1}}{\sum_{j=1}^{I-1} \delta l_j} \cdot S_k^J \quad (2)$$

Next, we consider the dependency of the tasks (or workers) in the dependency graph as shown in Figure 2. The more tasks that depend on task k, the higher priority that task k should have because its completion enables more other tasks to start running, which helps reduce JCT. In addition, if more dependent tasks are in layers closer to task k in the dependency graph, then it should have a higher priority because its completion enables more tasks to start earlier. Based on the rationale, we calculate the priority of task k considering ML features, $P_{k,J}^{ML}$, as follows:

$$P^{ML}_{k,J} = \begin{cases} P'^{ML}_{k,J}, & no\ dependent\ tasks \\ P'^{ML}_{k,J} + \gamma \sum_{i \in child(k)} P^{ML}_{i,J}, & otherwise \end{cases} \quad (3)$$

where $\gamma \in (0, 1)$ is a discounting factor to count the closeness between a dependent task and task k, $child(k)$ is the set of direct children of task k in the dependency graph. A larger γ means a higher weight is given to the priorities of a task's children when determining the task's priority. This priority determination method can be directly applied to the all-reduce communication structure to determine the priority of each task running in a worker. For the parameter server communication structure which has a separate centralized parameter server, its computation is also considered as a task and assigned with the highest priority since only after the parameter server is determined, the tasks in the workers know where to send their results.

For the computation features of task k of job J, we consider its task deadline ($d_{k,J}$), remaining running time ($r_{k,J}$) and waiting time in the queue ($w_{k,J}$). We consider two cases to estimate the remaining running time. If a job is executed previously, the deadline of each of its tasks can be calculated based on the job's deadline, dependency graph and historical task running time, as in [21]. We use the historical task running time as the task's required running time. If a job is not executed previously [25], we use sample running to estimate the required running time of each of the job's tasks and then infer the deadline of each task. Then, a task's remaining running time is calculated by $r_{k,J} = t_{k,J} - p_{k,J}$, where $t_{k,J}$ is the estimated required running time of task k and $p_{k,J}$ is the actual running time of task k. A task with a closer deadline, less remaining running time, or longer waiting time should have a higher priority to run because it helps meet the job deadline and improve the JCT. Considering the computation features, we calculate the priority of task k that does not have dependent tasks by:

$$P'^{C}_{k,J} = \gamma_d \cdot \frac{1}{(d_{k,J} - t)} + \gamma_r \cdot \frac{1}{r_{k,J}} + \gamma_w \cdot w_{k,J} \quad (4)$$

where t is the current time and γ_d, γ_r and γ_w are the weights of the considered factors. Larger γ_d, γ_r and γ_w lead to more weights on deadline consideration, job remaining time and job waiting time in the queue, respectively. The priority of task k the traditional computation features, $P^{C}_{k,J}$, is calculated by:

$$P^{C}_{k,J} = \begin{cases} P'^{C}_{k,J}, & no\ dependent\ tasks \\ P'^{C}_{k,J} + \gamma \sum_{i \in child(k)} P^{C}_{i,J}, & otherwise \end{cases} \quad (5)$$

By combining the priorities based on the ML features and computation features (Equs. (3) and (5)), $P_{k,J}$ is calculated by:

$$P_{k,J} = \alpha P^{ML}_{k,J} + (1 - \alpha) P^{C}_{k,J},\ \alpha \in [0, 1] \quad (6)$$

where α is a weight factor and a larger α means that the ML job features have higher weights than the computation features in determining a job's priority. MLF-H uses the task priority to order the tasks in the queue. Because tasks with higher priority are allocated to servers and executed earlier, it helps achieve our goal in Formula (1). Compared to previous job schedulers (e.g., Graphene [20]) that only consider traditional computation features and task dependency, MLFS is novel in that it additionally considers ML job spatial/temporal features in the priority determination. Also, it additionally considers the goal of increasing ML accuracy, which is not considered in previous schedulers for general jobs.

3.3.2 Basic Job Scheduling. MLF-H inserts newly submitted tasks and preempted tasks to the queue based on the priority. When there are underloaded servers and waiting tasks in the queue, it picks tasks one by one from the queue and assigns it to an underloaded node until there no more underloaded nodes or the queue is empty. Here, one problem is which node among the underloaded nodes should we choose to allocate each task from the queue? To solve this problem, we can rely on the method in [47] as described below.

We consider M types of resources (e.g., GPU, CPU, memory, and bandwidth), and more types of resources can easily be added. The utilization of type-m resource of server s at time slot t is calculated by $u^t_m = l^t_m / c_m$, where l^t_m is resource consumption of server s in time slot t and c_m is the capacity of server s on type-m resource. The utilization of type-m resource of task k is calculated by $u^t_{m,k} = l^t_{m,k} / c_m$, where $l^t_{m,k}$ is resource consumption of task k in time slot t. The resource utilization of server s at time t (U^t_s) is represented as a vector $(u^t_{1,s}, u^t_{2,s}, ..., u^t_{m,s}, ..., u^t_{M,s})$, and the resource utilization of task k at time t (U^t_k) is represented as a vector $(u^t_{1,k}, u^t_{2,k}, ..., u^t_{m,k}, ..., u^t_{M,k})$. We consider that type-$m$ resource in a server is overloaded if its u^t_m is higher than a pre-defined threshold h_r (e.g., 90%). A larger h_r helps more fully utilize the resources but increases the probability that a server becomes overloaded and hence JCT. Lower h_r increase the number of unnecessary task migrations. When at least one type of resources in a server are overloaded, we consider that this server is overloaded; otherwise, it is underloaded.

When we choose a server from several underloaded servers to allocate a task k, we hope that the chosen node is less likely to be overloaded, the communication bandwidth consumption between this task and other tasks is maximally reduced, and the task's performance degradation caused by the task movement to the node [10] is minimized. Avoiding overloaded servers and fully utilizing resources can help improve JCT and accuracy of the jobs in the system. To achieve these objectives, among the underloaded servers, we first determine ideal virtual host server for the task from the queue represented by $U^t_V = (u^t_{1,V}, u^t_{2,V}, ..., u^t_{m,V}, ..., u^t_{M,V}, u_{BW,V}, q_{k,V})$, in which $u^t_{m,V}$ is the minimum type-m resource utilization among all of the underloaded severs, $u_{BW,V}$ denotes the maximum value among the communication data sizes between the task and each of the underloaded servers (in order to allocate high-volume communicating tasks to the same server), $q_{k,V}$ is the task performance degradation caused by the task movement [10] and its ideal value is 0. The server that is most similar as the ideal virtual host server based on the Euclidean distance should be chosen. That is, the server with $\min ||U^t_s - U^t_V||$ among the candidate servers and will not be overloaded (on each resource and its least-loaded GPU) by hosting the task is selected as the host server for the task. Then, we schedule the task to the least-loaded GPU in the selected server. After one task is allocated, the scheduler continues to pick up the next task in the queue and schedule it. The scheduling stops when there are no more underloaded servers or the queue is empty. Finally, based on the determined task allocation schedule, the tasks are moved to their allocated GPUs.

3.3.3 Handling Overloaded Servers. MLF-H can be executed periodically or when there are underloaded nodes (and waiting tasks) or overloaded nodes. When there is an overloaded server, MLF-H chooses tasks in the overloaded server to be migrated out to relieve its excess load, inserts the tasks to the waiting queue, and reschedules the tasks along with the waiting tasks in the queue in the same manner. Note that these chosen migration tasks are virtually instead of actually moved to the queue in order to save the migration overhead. After one migration task is scheduled, it is directly moved from the overloaded server to the scheduled server. When there are no more underloaded servers but some migration tasks are still not assigned to servers, then these migration tasks are moved back to the queue.

Many previous ML job schedulers [21, 39, 53, 58] only determine task priority for task allocation (and preemption [21]) but do not handle server overload. Gandiva [55] handles GPU overload but does not consider other types of resources, though ML jobs also use other types of resources including CPU, memory, and bandwidth. Server overload on other resources still can adversely affect the ML job performance. MLF-H overcomes these problems. Below, we present how MLF-H selects migration tasks from an overloaded node while avoiding resource fragmentation to more fully utilize resources in the system.

For an overloaded server, to move out some of its current tasks to relieve its excess workload, an important question is which tasks we should choose to move out. To fully utilize resources and relieve the load on overloaded resources in an overloaded server, we use the method in [47]. That is, we first determine ideal virtual task to move out represented by $U_v^t = (u_{1,v}^t, u_{2,v}^t, ..., u_{m,v}^t, ..., u_{M,v}^t, u_{BW,v})$, in which $u_{m,v}^t$ for each overloaded resource is the maximum value among the tasks in the sever and $u_{m,v}^t$ for each underloaded resource is the minimum value among the tasks in the sever. $u_{BW,v}$ denotes the ideal communication data size between the migration task and existing tasks in the server and it equals to 0, which means that the task migration will not cause additional communication cost. Then, the task whose resource utilization is the closest to U_v^t based on the Euclidean distance, i.e., the task with min $||U_k^t - U_v^t||$ is selected to move out. If the server is still overloaded after it migrates out this selected task, the same process is repeated until the server is not overloaded. Here, we further advance this method by considering the ML features. First, we need to make sure that the tasks with high priorities will not be selected to migrate out in order to improve JCT and accuracy. Recall that the priority is calculated considering the accuracy improvement so the tasks that will contribute more on improving the accuracy should be less likely to be chosen to migrate out. Second, because ML tasks run in multiple GPUs in a server, each GPU must not be overloaded. Therefore, if there exist overloaded GPUs, we order the tasks in the overloaded GPUs based on the ascending order of the priority, and select tasks using the above method only among a certain percentage (p_s) of the tasks on the top until there are no overloaded GPUs. A smaller p_s means that the migration tasks have lower priorities, which constrains the performance degradation on JCT and accuracy but may not relieve overload quickly. When there are no overloaded GPUs, we select the task from all the running tasks in the server using the above method.

Stragglers may occur due to failing hardware, software bugs, misconfiguration and so on. To handle this problem, we can duplicate a task and assign the replica task to another server in task scheduling. Then, we use the output of the task that completes first and stop the other tasks. The number of replicas of a task depends on the possibility of the straggler occurrence. More replicas can better avoid straggler occurrence but generate more overhead. We leave detailed study of stragglers as future work.

3.4 ML Feature Based RL Task Scheduling (MLF-RL)

The heuristic MFL-H and other heuristic scheduling methods may not be able to fully catch ML job features or set optimal parameter values (e.g., γ, α), where obtaining the optimal schedule for goal in Equation (1) may be difficult. Also, the decision making in MFL-H may take a long time. To handle these problems, we rely on the deep RL technique. That is, MLFS initially runs MLF-H for a certain time period and uses the data to train MLF-RL, and then switches to MLF-RL when it is well trained. MLF-RL is novel compared with the previous RL-based job schedulers [35, 37, 39] in that it additionally considers ML features to improve both JCT and accuracy while previous RL-based job schedulers do not aim to improve accuracy or consider ML features.

An RL consists of an agent, state, action, and reward [50]. In each time step t, the agent observes the environment state s_t and chooses an action a_t based on its optimal policy in response to the current state and receives reward r_t. Recently, the Deep Neural Network (DNN) has become a popular function approximator because it automatically extracts features when solving large-scale RL problems [48]. Thus, we use DNN to serve as the agent, which generates the optimal policy. The output of DNN is the probability distribution of actions $\pi : \pi(s_t, a_t) \rightarrow [0, 1]$, where $\pi(s_t, a_t)$ denotes the probability of taking an action a_t at the current state s_t. The goal of the agent is to maximize the expected cumulative discounted reward: $\mathbb{E}[\sum_{t=0}^{\infty} \eta^t r_t]$, where $\eta \in (0, 1]$ is a factor discounting future rewards. A larger η enables the RL agent to consider more weights on the future rewards when updating the model.

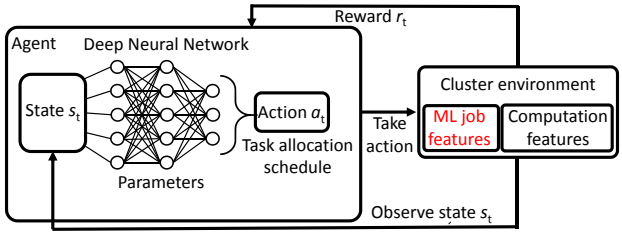


Figure 3: Deep RL structure in MLF-RL.

As shown in Figure 3, the state includes the information of all the waiting and running tasks and nodes in the cluster including the information needed to derive the ML job features and computation features used in MLF-H and additional information such as the ML algorithm name and dependency graph. More specifically, the state (or the RL input) includes: i) the information of tasks, including whether it is queuing or running, its job, arrival time, resource demand, its mini-batch, waiting time and running time; ii) the information of each task's job, including its ML algorithm, urgency level, deadline, number of maximum and finished iterations, loss

reduction for each finished iteration, mini-batch size, training data size, and dependency graph; and iii) the information of servers and nodes (GPUs), including the utilization of each resource type in a server, the utilization of each GPU, and current running tasks.

The DNN observes the state from the cluster environment and outputs the optimal action (i.e., task allocation schedule), represented by $A = \{(k, n)|k \in \mathcal{T}, n \in \mathcal{N}\}$. Recall that $\mathcal{N}$ also includes the waiting queue. The action includes the selection of tasks in overloaded nodes to move out and the assigned node (either underloaded node or queue) for each task in the selected migration tasks and waiting tasks in the queue.

The RL model has only one objective function to optimize, and a normal way to create the objective function for multiple objectives is to create a weighted sum of the objectives according to the approach in [35]. Therefore, to achieve our goal indicated in Formula (1), we define the reward function at time step t as follows:

$$r_t = \beta_1 g_1(A) + \beta_2 g_2(A) + \beta_3 g_3(A) + \beta_4 g_4(A) + \beta_5 g_5(A), \quad (7)$$

where β_i is the reward weight for the i^{th} $(i = 1, 2, ..., 5)$ objective according to Equation (1). A larger β_i $(i = 1, 2, ..., 5)$ value means a higher weight on the i^{th} objective. A problem here is how to determine the weight combination $(\beta_1, \beta_2, \beta_3, \beta_4, \beta_5)$ that generates better performance of RL's learned optimal policy in each of the objective dimensions. For this purpose, we could directly adopt the reward tuning method in [36] that uses Bayesian optimization to search for the weight combination. However, Bayesian optimization can only give better (instead of the best) results by exploring the whole weight search space and its time overhead is high [54]. After a limited number of rounds of Bayesian optimization, it cannot give fine-tuned weight results [36]. Specifically, Bayesian optimization cannot efficiently find the best results within a smaller value range [31]. Therefore, in order to obtain a better weight combination, we first run a limited number of rounds (e.g., 10) to obtain results, and then empirically try different combinations by slightly varying each value in the results. Finally, we choose the weight combination that generates the highest r_t in Equation (7).

Although the actual reward of the current scheduling decision can be known only when the scheduled tasks are completed, as in [35, 37], we wait for a time period after the scheduling decision is made at time t_0. That is, we compute the cumulative reward from t_0 to $t_0 + t_m$ as the reward of scheduling decision at time t_0: $\sum_{i=0}^{m} \eta^i \times r_{t_i}$. Note that we do not consider task migration overhead in the reward function here and leave it as our future work.

The DNN uses all of the aforementioned information and cumulative discounted reward to train the neural network, i.e., to update its policy neural network parameters θ to improve the scheduling decisions. We utilize the gradient-descent to update θ. The details of the DNN and the DNN training can be found in [51]. Only after the RL model is well trained (i.e., converged), MLFS switches from MLF-H to MLF-RL in order to output optimal scheduling decisions.

3.5 ML Feature Based System Load Control (MLF-C)

There is a tradeoff between ML model accuracy and ML job running time (the number of iterations) or the resource consumption [44]. To handle the ML overfitting problem, currently, users usually define a maximum epoch (i.e., iteration) to stop training or observe the validation loss curve to decide at which epoch the model performs the best and stop training. However, the minimum accuracy loss may appear earlier than the specified maximum epoch and then computation resources are wasted and JCT is increased by continuing training the model. Also, the human observation method relies on intuition and needs expert knowledge, which is not applicable for general users. Therefore, the proper time to stop training an ML model is when the maximum accuracy is obtained before the job deadline. For this purpose, we can use an early training stopping method in [17]. That is, when a job is running, we first use a weighted probabilistic learning curve model to predict the job's accuracy at the specified maximum iteration. If the predicted accuracy is less than an accuracy threshold, the training stops when the prediction confidence is higher than a threshold. Otherwise, the training continues and stops when the achieved accuracy reaches the accuracy threshold. The weighted probabilistic learning curve model is built based on historical data to fit one weighted learning curve for multiple types of ML jobs using Bayesian optimization. The inputs of the model include the number of iterations executed, the accuracy change for each executed iteration, the iteration when accuracy needs to be predicted, and its output is the job's predicted accuracy at the indicated interation. When one ML job is running, we monitor the accuracy change in real time.

In our proposed MLF-C, when users submit their ML jobs, they are asked to choose one of the following options: i) jobs run for the number of iterations indicated or controlled by them (i.e., the current approaches), ii) choose an ML iteration stopping algorithm (OptStop) (e.g., in [17]) that stops ML running when the achieved accuracy equals or is close to the maximum accuracy, or iii) only achieve the required accuracy. The users are also asked to indicate which of their chosen options can be changed by the system when the system is overloaded. For example, users choosing option i) allow the system to switch their choices to option ii) or iii) and users choosing option ii) allow the system to switch their choices to option iii) in order to help further reduce the system workload, which in turn improve their job JCT and accuracy performance. Further, the users are informed that option i) may lead to longer JCT especially when the system is overloaded. Based on the users' different choices and their performance requirements on accuracy and JCT, they are charged with different payment rates. In a private cluster, the system administrator can directly make the decision about the above options. When there are enough resources in the cluster, we can run the jobs based on user preference. When the system is overloaded, we will make changes on the user choices if the changes help reduce the system workload.

Thus, when the system is not overloaded, MLF-C follows the user choices accordingly, and when the system is overloaded, MLF-C changes the choices based on the users' indications to reduce system workload. Comparing to option i) (the current approaches), options ii) and iii) reduce the number of iterations of the ML jobs and hence system workload. As a result, it reduces JCT since jobs do not need to run more iterations or wait in the queue for a long time due to system overload while still providing near-optimal accuracy or meeting users' accuracy requirements. In addition, the jobs' accuracies by the job deadlines are improved since important iterations have a higher chance to run and will not be blocked due to the

running of unimportant iterations. As a result, both JCT and accuracy performances are improved. Therefore, when the system is not overloaded, MLF-C proactively helps avoid system overload, and when the system is overloaded, MLF-C reactively reduces system workload to mitigate the overload. In addition, MLF-C also helps people who do not have the knowledge about how many iterations are needed to achieve the maximum or their designed accuracy.

Next, we explain how we judge whether the system resources are limited or the system is overloaded. Recall that the resource utilization of server s at time t is represented as a vector $U_s^t = (u_1^t, u_2^t, ..., u_m^t, ..., u_M^t)$. The resource utilization of the cluster is represented as $U_c^t = (U_1^t, U_2^t, ..., U_{|N|}^t)$, where N is the set of servers in the cluster. The overload degree of server s is calculated by $O_s^t = ||U_s^t||$. The overload degree of the whole cluster is measured by the average of overload degrees of all servers: $O_c^t = \frac{1}{|N|}\sum_{s\in N}||U_s^t||$. The system is considered to be overloaded when there are tasks in the queue or when $O_c^t > h_s$, where h_s is a pre-defined threshold. A larger h_s means the more severe resource competition between jobs leads to higher JCT. However, a lower h_s may cause a higher overhead for system load control. In this case, as mentioned above, based on the users' indications, MLF-C makes changes on user selected options if the changes reduce the number of iterations, and stops producing or running unnecessary tasks accordingly. Consequently, the system does not need to further generate or run tasks for the next iteration for the job once the selected option's accuracy requirement is met.

4 PERFORMANCE EVALUATION

4.1 Experiment Settings

Real trace: We used a publicly available trace of DNN training workloads from Microsoft [3] to run trace-driven simulation. The trace contains a representative subset of the first-party DNN training workload on Microsoft's internal Philly clusters with 550 servers and 2474 GPUs collected from Aug. 07, 2017 to Dec. 22, 2017. The trace has 117325 jobs including Convolutional Neural Networks (CNNs), LSTMs and RNNs. It contains two types of information: 1) for each job, it contains the job arrival time, the number of GPUs requested, GPU allocation position, job completion status (the highest accuracy value when the job finished), and the job running information reported per minute including CPU, memory, and GPU utilization; and 2) for each server, it contains the CPU, memory, and GPU utilization per minute [26]. In our experiment, we use the job arrival time, the number of GPUs requested and job completion status as the accuracy requirement of each job.

These ML algorithms include AlexNet, ResNet, MLP, LSTM and Support Vector Machine (SVM). In SVM, we only used data parallelism. In MLP and AlexNet, because of their sequential task dependency graph structures, we partitioned the model sequentially into several parts for model parallelism and also used data parallelism. In LSTM and ResNet, we used both data parallelism and model parallelism and partitioned each layer into several parts for model parallelism. Therefore, we used mixed workloads in our experiments. The batch size is 1MB for AlexNet and ResNet, and 1.5KB for LSTM, MLP and SVM. To create each ML job, we randomly selected one of the ML algorithms. The size of the training data is randomly selected from [100,1000]MB. We scaled down the number of jobs as in [21] that used 60 GPUs and 480 jobs in experiments. We used 80 GPUs. The duration of the trace is 18 weeks, and we randomly selected one week to do the test.

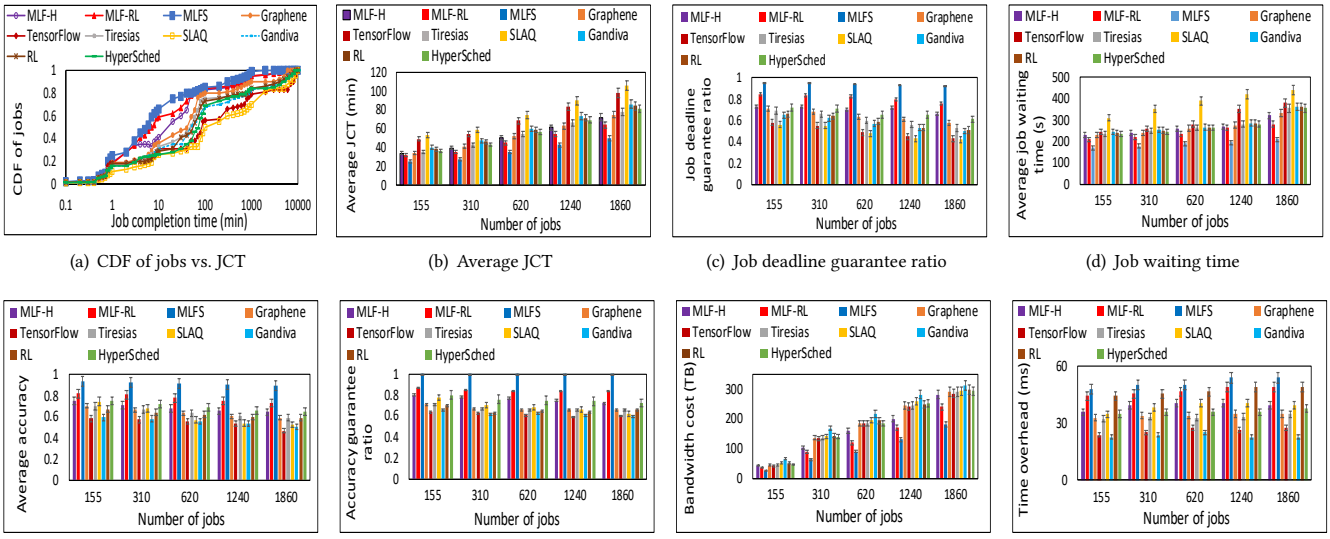
Real implementation: We conducted Pytorch [4] based implementation on Amazon AWS [40] for our methods and comparison methods. We used 20 p3.8xlarge instances (to represent servers), which form a 80-GPU cluster [21], where each server has 4 Nvidia Tesla V100 GPUs (16 GB RAM, 5120 CUDA Cores and 640 Tensor Cores), 32 vCPU cores (based on Intel Xeon E5-2686 v4 processor), and 244 GB memory [1]. We used the job arrival time from the real trace [3], as well as the training data and five ML algorithms used in [55] (downloaded from GitHub [2, 6]) with their default batch size. We set the number of jobs to $620x$, where x equals to 1/4, 1/2, 1, 2, and 3.

Simulation: We developed a Python based simulator (running on Google Colab platform) for large-scale testing with 2474 GPUs in 550 servers. We set the number of jobs to $117325x$ jobs, where x equals to 1/2 and then is varied from 1 to 4 with 1 increase in each step. We used all trace data to drive the simulation. Given that the trace does not contain bandwidth cost information, for each job, we randomly selected two values within [50,100]MB as the communication volume between each worker to the parameter server and between workers in one communication since [21] indicates that the communication volume is the same across iterations.

As we use both data parallelism and model parallelism, we need the corresponding information of the tasks in this scenario to drive our simulation. However, the trace is not from the data parallelism and model parallelism scenario nor provides the name of the ML algorithms. We then first sample-ran the aforementioned five ML algorithms with the aforementioned parallelism methods using different number of GPUs and recorded the job completion times. Then, we mapped each job in the trace to one of our running ML algorithms with a certain number of GPUs that has similar job latency. Next, we ran the ML algorithm with our data parallelism degree and different model parallelism degrees to get the information of different tasks in the job in this scenario.

Experimental setting: In the experiments, we considered resource types including CPU, memory, GPU and bandwidth cost. The number of GPUs needed by each ML job was randomly selected from {1, 2, 4, 8, 16, 32}. We also set the number of model partitions to this number. SVM did not run in model parallelism because it is hard to partition its network model. The default parameter values of our method are listed below: α=0.3, γ=0.8, γ_d=0.3, γ_r=0.3, γ_w=0.35, $\beta_1 = 0.5$, $\beta_2 = 0.55$ (larger β_2 means more weights on deadline guarantee), $\beta_3 = 0.25$, $\beta_4 = 0.15$, $\beta_5 = 0.15$, η=0.95, $h_r = h_s$=90%, and p_s=10%. We chose these default values because in our experiment, we found that these parameters can achieve best performance in our evaluation environment. In practice, these tunable parameters of a cluster are determined by the administrator of the cluster according to the goals they concern and the particular cluster environment and configurations.

The job scheduler runs every minute. For each job, we randomly used $\max\{1.1t_e, t_r\}$ as its deadline, where t_e is its estimated execution time and t_r was randomly chosen from $[\frac{1}{2}, 24]$hours. Unless otherwise specified, the number of iterations run by each job is the same as that in the trace. In the RL training process in MLF-RL, we randomly scheduled tasks to nodes and calculated reward, and



(a) CDF of jobs vs. JCT (b) Average JCT (c) Job deadline guarantee ratio (d) Job waiting time

(e) Average accuracy (f) Accuracy guarantee ratio (g) Bandwidth cost (h) Scheduler overhead

Figure 4: Overall performance in real experiments.

then updated the RL model. After the RL processed the first 50% data of the real trace, the model is trained which takes around 26 hours. The error bar in our experimental figures represents the 1^{th} and 99^{th} percentiles and median of the result values from 10 experiments. We use $\frac{y-z}{z}$ to calculate the improvement when comparing the performance of method y and z.

Comparison methods: We compared our methods with the state-of-the-art ML job schedulers including SLAQ [58], Gandiva [55], Tiresias [21], RL [39], Graphene [20], TensorFlow [8] that uses the Borg resource manager [53] and HyperSched [32]. RL aims to minimize the JCT and it uses data parallelism and partitions the model sequentially into several parts for model parallelism. The number of parts equals the number of layers in one job. The details of these methods are explained in Section 2. We implemented SLAQ, RL and Tetris by ourselves and used the open source codes of other comparison methods.

In MLF-C, we assume that all jobs use OptStop and it stops a job when its required accuracy is reached when the system is overloaded. We open-sourced our code in Github [5].

4.2 Experimental Results

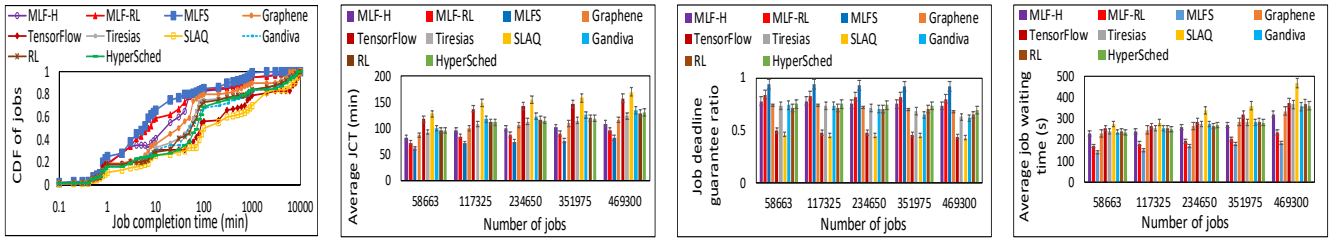
4.2.1 Overall Performance Comparison. Figure 4 and Figure 5 compare the overall performance of our methods against other methods in real experiments and simulation, respectively. Given that both sets of figures show similar trends and orders, we discuss the two figures together and show the performance improvement ratios from the simulation in "()" in the following.

Figures 4(a) and 5(a) show the cumulative distribution function (CDF) of JCT of each method. The results show that the overall JCT follows: MLFS<MLF-RL<MLF-H<Graphene<Tiresias≈Hyper Sched≈RL≈Gandiva<TensorFlow<SLAQ. We use $\prec$ to mean slightly lower relationship and use $\succ$ to mean slightly higher relationship. Over 85%, 81%, and 78% jobs in MLFS, MLF-RL and MLF-H have JCTs less than 100 minutes, respectively, in real experiments. Considering the percentage of jobs with JCTs less than 100 minutes, MLF-RL outperforms MLF-H by 4% (4%), and MLFS improves MLF-RL by 5% (6%) due to additional MLF-C. Meanwhile, 61%, 60%, 60%, 55%, 46%, and 39% jobs have JCTs less than 100 minutes in Tiresias, HyperSched, RL, Gandiva, TensorFlow and SLAQ, respectively. MLFS improves Tiresias by 33% (38%) and improves SLAQ by 118% (128%).

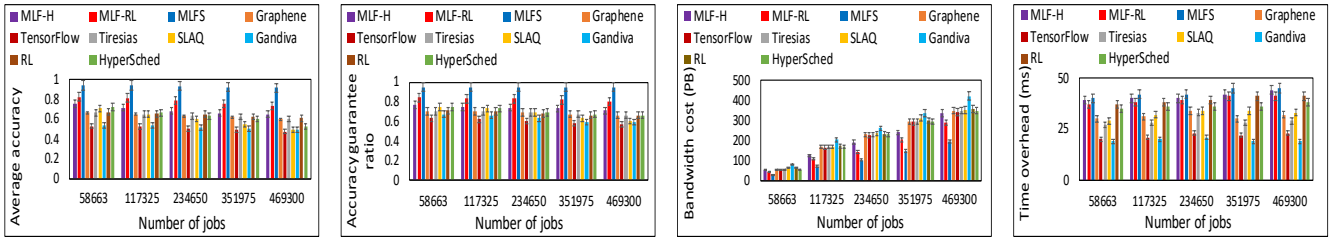
Both MLF-RL and MLF-H jointly consider both temporal/spatial ML job features and computation features in scheduling tasks to the underloaded servers and also in selecting migration tasks from overloaded servers. The additional consideration of ML job features compared to other methods helps them achieve lower JCT. Since the tasks in earlier iterations that have more closer dependent tasks in the dependency graph have higher priorities to run, their completion enables more tasks to start earlier, thus helping reduce JCTs. In addition, MLF-RL and MLF-H consider computation features related to JCT which directly reduces JCTs. MLF-RL outperforms MLF-H because MLF-RL can better extract ML job features, adapt to learn the scheduling policy and output the optimal schedule whereas MLF-H may not be able to set optimal parameter values for its parameters.

MLFS produces significantly lower JCT than other methods. The reduction of JCT is contributed from MLF-RL and MLF-C. MLF-C stops training ML jobs at their optimal stopping epochs to save computation resources and time without degrading their accuracy performance and also stops tasks in later iterations when the desired accuracy is reached to avoid resource overload and task waiting, thus reducing JCT.

In order to reduce JCT, Graphene considers the task dependancy graph. The tasks with more dependent tasks have higher priority to be scheduled. The earlier completion for this kind of tasks leads to the earliest start of other dependent tasks and then decreased JCT. Thus, Graphene achieve better JCT reduction performance compared with other comparison methods. Tiresias gives the jobs that can complete in the next service epoch the highest priority



(a) CDF of jobs vs. JCT (b) Average JCT (c) Job deadline guarantee ratio (d) Job waiting time



(e) Average accuracy (f) Accuracy guarantee ratio (g) Bandwidth cost (h) Scheduler overhead

Figure 5: Overall performance in large-scale simulation.

to run and RL uses the RL technique for the job scheduling, but they do not consider the ML features. HyperSched, Gandiva and TensorFlow do not directly aim to improve JCT. HyperSched aims to increase the accuracy of jobs within their deadlines. Gandiva uses FIFO and also migrate tasks between GPUs to fully utilize GPU resources, and TensorFlow uses Fair scheduler that aims to achieve the fairness between jobs in resource allocation. As a result, HyperSched, Tiresias and RL produce JCTs higher than our methods, but lower than TensorFlow. Gandiva's JCT is comparable to those of HyperSched, Tiresias and RL, and lower than TensorFlow due to Gandiva's additional task migration. SLAQ only aims to maximize the accuracy improvement across jobs rather than JCT, thus it produces higher JCT than TensorFlow. In conclusion, in terms of the JCT, MLFS outperforms other methods and our proposed methods MLF-H, MLF-RL and MLF-C are all effective in reducing JCT.

Makespan is the time period from when the first job is submitted to when the last job is completed. The makespan is 40-90 (63-336) hours in MLFS, 51-102 (67-342) hours in MLF-RL, and 54-116 (72-349) hours in MLF-H. For the highest workload, MLFS improves MLF-RL by 10% (9%) and MLF-RL improves MLF-H by 15% (11%). Also, the makespan is 56-125 (74-355) hours in Tiresias, 58-128(78-360) hours in HyperSched, 61-134 (78-361) hours in RL, 64-138 (84-368) hours in Gandiva, 78-158 (103-416) hours in TensorFlow, and 85-170 (11-415) hours in SLAQ. Thus, MLFS improves Tiresias by 32% (17%) and improves SLAQ by 52% (39%).

Figures 4(b) and 5(b) show the average JCT for each method. We see that the result follows: MLFS<MLF-RL<MLF-H<Graphene<Tiresias≈HyperSched≈RL≈Gandiva<TensorFlow<SLAQ. For 1860 jobs, MLFS improves MLF-RL by 22% (18%) (due to additional MLF-C) and MLF-RL improves MLF-H by 11% (10%). Meanwhile, MLFS improves Tiresias by 34% (30%) and improves SLAQ by 53% (47%.). The results show similar orders and trends of all methods as in Figures 4(a) and 5(a) due to the same reasons.

Figures 4(c) and 5(c) show the job deadline guarantee ratio, which is the percent of the jobs whose deadlines are satisfied. We see that the result follows: MLFS>MLF-RL>MLF-H>HyperSched >Graphene>Tiresias≈RL≈Gandiva>TensorFlow>SLAQ. For 1860 jobs, MLFS improves MLF-RL by 21% (16%) and MLF-RL improves MLF-H by 16% (21%). Meanwhile, MLFS improves HyperSched by 47% (43%) and improves SLAQ by 102% (101%). Part of the reasons for the results are the same as those for Figures 4(a) and 5(a). HyperSched considers the job deadline and tends to give more resources to the tasks with the best potential accuracy performance before the deadlines. These methods try to satisfy job deadlines in job scheduling while other methods do not.

A job's waiting time is the accumulated time periods in which none of its tasks are running from its submission time to its completion time. Figures 4(d) and 5(d) plot the average job waiting time per job. The result shows that MLFS<MLF-RL<MLF-H<Graphene <HyperSched≈Tiresias≈RL≈Gandiva<TensorFlow<SLAQ. ML FS improves MLF-RL by 25% (22%) (due to additional MLF-C) and MLF-RL improves MLF-H by 14% (21%). MLFS improves Tiresias by 40% (47%) and improves SLAQ by 52% (57%). Part of the reasons are similar as those in Figures 4(a) and 5(a) since a lower JCT enables more jobs to run earlier and helps reduce job waiting time. In addition, waiting time is a factor considered in MLF-RL and MLF-H in priority determination in order to reduce the task waiting time, and MLF-C further reduces job waiting time by removing some unnecessary tasks. Though HyperSched pauses jobs that do not increase accuracy significantly, it does not aim to reduce JCT, so jobs are always running or waiting before their deadlines. As a result, our methods generate lower waiting time than other methods, and MLFS produces the least average job waiting time.

Figures 4(e) and 5(e) plot the average accuracy of ML jobs by their deadlines. We see that the result follows: MLFS>MLF-RL>MLF-H>HyperSched>SLAQ>Tiresias≈RL≈Graphene>Gandiva≈Tensor Flow for the lowest workload, and MLFS>MLF-RL>MLF-H>Hyper Sched>Tiresias≈RL≈ sorFlow for high workload. For 155 jobs, MLFS improves MLF-RL by 10% (11%) and MLF-RL improves MLF-H by 7% (8%). Also, MLFS improves Tiresias by 14% (16%) and improves TensorFlow by 44%

(46%). For 1860 jobs, MLFS improves MLF-RL by 23% (20%) and MLF-RL improves MLF-H by 14% (8%). Also, MLFS improves HyperSched by 43% (41%) and improves TensorFlow by 64% (60%).

On the one hand, with a lower workload in the cluster, HyperSched performs the best among the comparison methods since it pauses jobs that do not gain high accuracy increase and also gives higher priorities in resource allocation to the task which tends to achieve the best accuracy performance before its deadline. Tiresias, RL and Graphene try to improve JCT so that the jobs can complete earlier, which also increases accuracy by the job deadlines due to resource competition mitigation. As a result, they achieve higher accuracy than SLAQ though SLAQ aims to improve accuracy. Gandiva and the Fair scheduler in TensorFlow do not directly aim to improve JCT or accuracy. Therefore, many jobs must wait and their accuracy by the deadline is lower than other methods. Since MLFS considers both accuracy and JCT requirements and leverages ML job features, it achieves the highest average accuracy in both high and low workloads.

Figures 4(f) and 5(f) show the accuracy guarantee ratio, i.e., the percentage of ML jobs whose accuracy requirements are satisfied by their deadlines. The trends and orders of result are similar as in Figures 4(e) and 5(e). For 155 jobs, MLFS improves MLF-RL by 15% (18%) and MLF-RL improves MLF-H by 20% (19%). Also, MLFS improves HyperSched by 36% (35%) and improves TensorFlow by 56% (52%). For 1860 jobs, MLFS improves MLF-RL by 22% (20%) and MLF-RL improves MLF-H by 8% (9%). Also, MLFS improves HyperSched by 32% (34%) and improves TensorFlow by 61% (54%). Part of the reasons for the results are the same reasons as in Figures 4(e) and 5(e). In addition, meeting the accuracy requirement is an objective in our methods, but is not considered in other methods. As a result, our methods produce the highest accuracy guarantee ratio.

Figures 4(g) and 5(g) plot the communication bandwidth cost. We see that the result follows: MLFS<MLF-RL<MLF-H≈TensorFlow ≈Graphene≈Tiresias≈HyperSched≈SLAQ≈RL<Gandiva. For 1860 jobs, MLFS improves MLF-RL by 22% (24%) and MLF-RL improves MLF-H by 14% (15%). Meanwhile, MLFS improves TensorFlow by 36% (37%) and improves Gandiva by 45% (54%). MLF-H tries to reduce bandwidth cost when selecting a node to allocate a task and when selecting tasks to migrate out of an overloaded node. MLF-RL considers bandwidth cost as a part of reward function. MLF-C further removes tasks that make little or no contribution to the desired accuracy. As a result, MLFS, MLF-H and MLF-RL produce less bandwidth cost than other methods, and MLFS produces the least bandwidth cost. Gandiva simply uses FIFO without trying to reduce JCT and uses task migration to improve resource utilization without considering bandwidth cost. Therefore, the task migration introduces extra bandwidth cost, so Gandiva generates the highest bandwidth cost. All other comparison methods do not aim to reduce bandwidth cost, thus producing higher bandwidth cost than MLF-H, MLF-RL, and MLFS.

We recorded a scheduler's running time for each scheduling and calculated its average value. Figure 4(h) plots the average time overhead of different schedulers versus different numbers of jobs. We see that the result follows: Gandiva<TensorFlow<Tiresias<Graphene <HyperSched<SLAQ<RL≈MLF-RL<MLF-H<MLFS. For 1860 jobs, the scheduler time overhead is 54ms, 49ms, and 40ms in MLFS, MLF-RL and MLF-H, and it is 23ms, 28ms, 35ms, 38ms, 40ms, and 49ms in

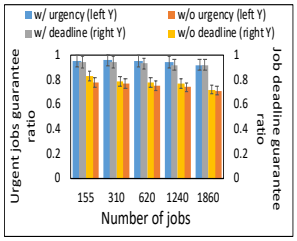


Figure 6: Urgency and deadline consideration.

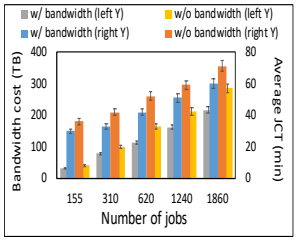


Figure 7: Bandwidth consideration.

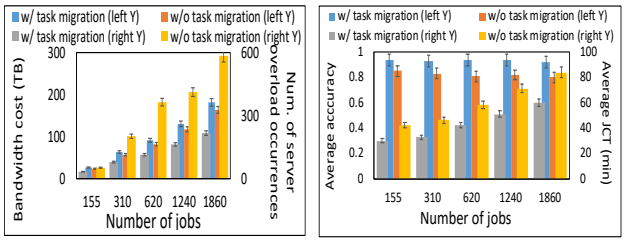
Gandiva, TensorFlow, Tiresias, HyperSched, SLAQ and RL, respectively. MLF-RL and RL use RL technique so that they have similar time overhead. MLFS generates higher time overhead since it has additional MLF-C in addition to MLF-RL. MLF-H consists two parts, a heuristic job scheduling method and handling overloaded servers, which needs to select migration tasks and destination nodes. Thus, it generates higher time overhead than MLF-RL and RL and less overhead compared with MLFS consisting of MLF-RL and MLF-C. The results indicate the advantages of MLF-RL and RL in reducing the time for decision making. SLAQ, Tiresias, Graphene and HyperSched use a single heuristic method without handling server overload or system overload, which cost less time overhead compared with the RL-based methods. Gandiva uses simple FIFO and TensorFlow uses the simple Fair scheduler, so they produce the smallest scheduler overhead. Though our methods have slightly higher time overhead in the *ms* level than other methods, they achieve much higher JCT and accuracy as shown in the above.

In all of the figures, we see that as the number of jobs increases, the average JCT, average job waiting and bandwidth cost increase. This is because more jobs generate higher workload and hence longer running time and waiting time, and higher bandwidth cost for the job running. One of the reasons that MLFS performs better in both JCT and accuracy than comparison methods is that it uses OptStop, which helps reduce JCT while still achieving the highest accuracy of the stopped jobs and leaves more resources for other jobs to achieve high accuracy.

4.2.2 Performance of Each System Component. We conducted real experiments to show the effectiveness of each system component of MLF-H and MLF-C.

Factors in Priority Determination. We set the urgency level of each job to a value randomly selected from [1,10] and consider the jobs with urgency level higher than 8 as urgent jobs. Figure 6 shows the deadline guarantee ratio for urgent jobs with and without urgency coefficient consideration in the priority calculation in Equ. (2). It shows that this consideration improves the deadline guarantee ratio by 22%-30%. Figure 6 also plots the job deadline guarantee ratio with and without job deadline consideration in Equ. (4). We find that the deadline consideration improves the job deadline guarantee by 13%-25%.

Figure 7 shows the average JCT and bandwidth cost with and without the bandwidth cost consideration in Equ. (2). We observe that the bandwidth cost consideration reduces the JCT by 5%-15% and reduces the bandwidth cost by 20%-35%, which confirms the importance of considering reducing bandwidth cost.



(a) Server overload occurrence (b) Average accuracy and JCT

Figure 8: Effectiveness of system load control.

Effectiveness of Task Migration. Recall that MLF-H has a task migration component to relieve excess load from overloaded servers. Figures 8(a) and 8(b) show the number of server overload occurrences, bandwidth cost, average accuracy by job deadline, and average JCT with and without task migration, respectively. We notice that the task migration component reduces the number of server overload occurrences by 36%-60% and increases the bandwidth cost by 10%-14%. The task migration also increases the average accuracy by 8%-10% and reduces the average JCT by 15%-24%. Server overload leads to longer waiting time and slower processing speed that lowers accuracy by the deadline. Task migration helps relieve excess load from overloaded servers.

Effectiveness of ML-based System Load Reduction. Figure 9 shows the accuracy guarantee ratio and average JCT with and without MLF-C (in Section 3.5). MLF-C improves the accuracy guarantee ratio by 17%-23% and the average JCT by 28%-42%. Server overload leads to longer waiting time and slower processing speed that lowers accuracy by the deadline. MLF-C solves this problem by removing unnecessary tasks when the desired accuracy is reached.

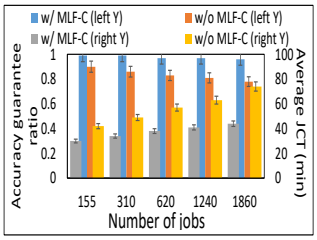


Figure 9: System load reduction.

5 SUMMARY AND LIMITATIONS

Compared to other ML cluster schedulers, first, MLFS is novel in that it combines the ML features and computation features together to achieve better performance in both JCT and accuracy. Second, the RL based method (MLF-RL) in MLFS improves both JCT and accuracy of the ML feature based RL task scheduling method (MLF-RL), thus contributing the JCT and accuracy performance of MLFS. Third, with system load control (MLF-C), MLFS generates higher time overhead but improves both JCT and accuracy. Fourth, comparing with other job schedulers, MLFS has two additional methods: task migration and system load control, which can significantly improve JCT and accuracy. Further, the improvement on JCT also leads to the improvement on accuracy since earlier job completion helps avoid resource competition and enables ML jobs to run more iterations before their deadlines.

There are several limitations of MLFS. First, MLFS needs task dependency graph to determine the priorities of different tasks. For an ML model that cannot be partitioned, our method may achieve similar performance as other methods. Second, when the RL job scheduler needs to be used in a new environment (e.g., when there is a new model or a new GPU type), the previous trained model under different scenarios may not be able to achieve the best performance because the different characters of ML jobs and the different computing ability of different hardware can degrade the performance of RL model. The system needs to first collect enough data in the new environment and then re-train the RL model to achieve the better performance. Third, our method only considers the bandwidth cost without considering the cluster network topology though the network bottleneck caused by switches or other network hardware may degrade the job performance. We leave handling of these limitations as future work.

6 CONCLUSION

With the rapid development of large-scale ML techniques, ML job scheduling has become increasingly important in recent years. To meet this critical need, we propose MLFS. Compared with previous ML job schedulers, the advantages of MLFS are that i) it intelligently leverages ML job features to significantly improve JCT and accuracy, ii) it can be applied for ML model parallelism, which is an approach for large-scale ML and DL jobs, and iii) it can meet both deadline and accuracy requirements of ML applications even when the system is overloaded. Considering the spatial/temporal ML features and computation features, MLFS determines the task priority for job scheduling to achieve our goal. MLFS also uses deep RL in task scheduling to achieve the goal by considering all the features. In addition, MLFS intelligently controls the system load by removing tasks once the desired accuracy is reached when the system is overloaded and determines the optimal number of iterations that an ML job runs to maximize accuracy while avoiding running more iterations. Both trace-driven large-scale simulation and real implementation show that MLFS has superior performance in JCT and accuracy than state-of-the-art ML job schedulers, and the effectiveness of each component in MLFS.

This work is the first to explore how to leverage ML job features to improve ML job schedulers, and we hope that it can stimulate more research work in this direction. In the future, we will explore more features of the data parallelism and model parallelism scenario and leverage the features in job scheduling. We will also study how to adaptively determine the number of iterations of the ML jobs in the system based on available resources to maximize accuracy. In addition, we will explore approaches to handle stragglers in the data parallelism and model parallelism scenario. Further, we will study the sensitivity of the parameters in MLFS and how to enable MLFS to handle the case that when there is a new model or a new GPU type.

ACKNOWLEDGEMENTS

We thank the valuable suggestions from the anonymous reviewers and the shepherd. This research was supported in part by U.S. NSF grants NSF-1827674, CCF-1822965 and Microsoft Research Faculty Fellowship 8300751, and AWS Machine Learning Research Awards.

REFERENCES

[1] [Accessed in June 2020]. Amazon EC2 types. https://aws.amazon.com/ec2/instance-types/.
[2] [Accessed in June 2020]. ILSVRC2010. https://github.com/Abhisek-/AlexNet.
[3] [Accessed in June 2020]. Microsoft DNN trace. https://github.com/msr-fiddle/philly-traces/.
[4] [Accessed in June 2020]. PyTorch. https://pytorch.org.
[5] [Accessed in June 2020]. Source Code. https://github.com/hiddenlayer2020/ML-Job-Scheduler-MLFS.
[6] [Accessed in June 2020]. Text classification on R8 Dataset. https://github.com/jiangqy/LSTM-Classification-Pytorch.
[7] [Accessed in Oct. 2020]. Baidu, Ring all reduce.. https://github.com/baidu-research/baidu-allreduce.
[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard. 2016. Tensorflow: A system for large-scale machine learning. In *Proc. of OSDI*.
[9] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. 2015. Information-agnostic flow scheduling for commodity data centers. In *Proc. of NSDI*.
[10] A. Beloglazov and R. Buyya. 2012. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation: Practice and Experience*.
[11] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proc. of EuroSys*.
[12] C. Chen, C. Yang, and H. Cheng. 2018. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839* (2018).
[13] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. 2016. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *Proc. of NSDI*.
[14] A. Chung, W. Park, and R. Ganger. 2018. Stratus: cost-aware container scheduling in the public cloud. In *Proc. of SOCC*.
[15] C. Curino, S. Krishnan, K. Karanasos, S. Rao, M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, and S. Heddaya. 2019. Hydra: a federated resource manager for data-center scale analytics.. In *Proc. of NSDI*.
[16] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. 2018. Kairos: Preemptive data center scheduling without runtime estimates. In *Proc. of SOCC*.
[17] T. Domhan, J. Springenberg, and F. Hutter. 2019. Speeding up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves. *International Conference on Machine Learning* (2019).
[18] A. Gholami, A. Azad, P. Jin, K. Keutzer, and A. Buluc. 2018. Integrated Model, Batch, and Domain Parallelism in Training Neural Networks. In *Proc. of SPAA*.
[19] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. 2015. Multi-resource packing for cluster schedulers. In *Proc. of SIGCOMM*.
[20] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. 2016. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. *Proc. of OSDI* (2016).
[21] J. Gu, M. Chowdhury, K. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. *Proc. of NSDI* (2019).
[22] Y. Han, X. Wang, V. Leung, D. Niyato, X. Yan, and X. Chen. 2019. Convergence of Edge Computing and Deep Learning: A Comprehensive Survey. *arXiv preprint arXiv:1907.08349*.
[23] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. 2018. TicTac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288* (2018).
[24] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. Reiss. 2015. Resource elasticity for large-scale machine learning. In *Proc. of SIGMOD*.
[25] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. Le, and Y. Wu. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Proc. of NIPS*.
[26] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proc. of ATC*.
[27] Z. Jia, M. Zaharia, and A. Aiken. 2019. Beyond data and model parallelism for deep neural networks. In *Proc. of SysML*.
[28] M. Laumanns, L. Thiele, and E. Zitzler. 2006. An efficient, adaptive parameter variation scheme for metaheuristics based on the epsilon-constraint method. *European Journal of Operational Research*.
[29] T. Le, X. Sun, M. Chowdhury, and Z. Liu. 2020. AlloX: compute allocation in hybrid clusters. In *Proc. of EuroSys*.
[30] S. Lee, J. Kim, X. Zheng, Q. Ho, G. Gibson, and E. Xing. 2014. On Model Parallelization and Scheduling Strategies for Distributed Machine Learning. In *Proc. of NIPS*.
[31] B. Letham and E. Bakshy. 2019. Bayesian Optimization for Policy Search via Online-Offline Experimentation. *Journal of Machine Learning Research* (2019).
[32] R. Liaw, R. Bhardwaj, L. Dunlap, Y. Zou, J. Gonzalez, I. Stoica, and A. Tumanov. 2019. Hypersched: Dynamic resource reallocation for model development on a deadline. In *Proc. of SoCC*.
[33] Y. Lin, S. Han, H. Mao, Y. Wang, and W. Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*.
[34] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. 2015. Retro: Targeted resource management in multi-tenant distributed systems. In *Proc. of NSDI*.
[35] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. 2016. Resource management with deep reinforcement learning. In *Proc. of Hotnet*.
[36] H. Mao, S. Chen, D. Dimmery, S. Singh, D. Blaisdell, Y. Tian, M. Alizadeh, and E. Bakshy. 2019. Real-world Video Adaptation with Reinforcement Learning. In *Proc. of ICML*.
[37] H. Mao, M. Schwarzkopf, B. Venkatakrishnan, Z. Meng, and M. Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. *Proc. of SIGCOMM*.
[38] H. Mikami, H. Suganuma, Y. Tanaka, and Y. Kageyama. 2018. Massively distributed SGD: ImageNet/ResNet-50 training in a flash. *arXiv preprint arXiv:1811.05233* (2018).
[39] A. Mirhoseini, H. Pham, V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean. 2017. Device placement optimization with reinforcement learning. In *Proc. of ICML*.
[40] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, and M. Jordan. 2018. Ray: A distributed framework for emerging AI applications. In *Proc. of OSDI*.
[41] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, P. Gibbons, and M. Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. *Proc. of SOSP* (2019).
[42] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proc. of EuroSys*.
[43] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Meng, and W. Lin. 2019. DL2: A Deep Learning-driven Scheduler for Deep Learning Clusters. In *arXiv:1909.06040*.
[44] L. Prechelt. 1998. Early stopping-but when? In *Neural Networks: Tricks of the trade*.
[45] J. Rasley, Y. He, F. Yan, O. Ruwase, and R. Fonseca. 2017. Hyperdrive: Exploring hyperparameters with pop scheduling. In *Proc. of Middleware*.
[46] A. Sergeev and M. Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).
[47] H. Shen and L. Chen. 2017. RIAL: Resource intensity aware load balancing in clouds. *Trans. on Cloud Computing*.
[48] D. Silver, A. Huang, J. Maddison, A. Guez, L. Sifre, G. Van, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, and M. Lanctot. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*.
[49] P. Sun, Y. Wen, N. Ta, and S. Yan. 2017. Towards distributed machine learning in shared clusters: A dynamically-partitioned approach. In *Proc. of SMARTCOMP*.
[50] R. Sutton and A. Barto. 2014. Introduction to reinforcement learning. *MIT press*.
[51] S. Sutton, A. McAllester, P. Singh, and Y. Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Proc. of NIPS*.
[52] A. Tumanov, T. Zhu, J. Park, M. Kozuch, M. Harchol-Balter, and G. Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proc. of EuroSys*.
[53] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Pro. of EuroSys*.
[54] J. Wang, J. Xu, and X. Wang. 2018. Combination of hyperband and bayesian optimization for hyperparameter optimization in deep learning. *arXiv preprint arXiv:1801.01596* (2018).
[55] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, and Q. Zhang. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *Proc. of OSDI*.
[56] H. Yabuuchi, D. Taniwaki, and S. Omura. 2019. Low-latency job scheduling with preemption for the development of deep learning. In *Proc. of OpML*.
[57] J. Zhan, O. Kayıran, G. Loh, C. Das, and Y. Xie. 2016. OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures. In *Proc. of MICRO*.
[58] H. Zhang, L. Stafman, A. Or, and J. Freedman. 2017. Slaq: quality-driven scheduling for distributed machine learning. In *Proc. of SOCC*.