Deep Learning-based Job Placement in Distributed Machine Learning Clusters

Yixin Bao*, Yanghua Peng*, Chuan Wu*
*Department of Computer Science, The University of Hong Kong, Email: {yxbao,yhpeng,cwu}@cs.hku.hk

***Abstract*—Production machine learning (ML) clusters commonly host a variety of distributed ML workloads, *e.g.*, speech recognition, machine translation. While server sharing among jobs improves resource utilization, interference among co-located ML jobs can lead to significant performance downgrade. Existing cluster schedulers (*e.g.*, Mesos) are interference-oblivious in their job placement, causing suboptimal resource efficiency. Interference-aware job placement has been studied in the literature, but was treated using detailed workload profiling and interference modeling, which is not a general solution. This paper presents *Harmony*, a deep learning-driven ML cluster scheduler that places training jobs in a manner that minimizes interference and maximizes performance (*i.e.*, training completion time). *Harmony* is based on a carefully designed deep reinforcement learning (DRL) framework augmented with reward modeling. The DRL employs state-of-the-art techniques to stabilize training and improve convergence, including actor-critic algorithm, job-aware action space exploration and experience replay. In view of a common lack of reward samples corresponding to different placement decisions, we build an auxiliary reward prediction model, which is trained using historical samples and used for producing reward for unseen placement. Experiments using real ML workloads in a Kubernetes cluster of 6 GPU servers show that *Harmony* outperforms representative schedulers by 25% in terms of average job completion time.**

I. Introduction

Nowadays most leading IT companies operate machine learning (ML) clusters of GPU servers. Various ML workloads are run on the cluster, to support the company's services. For example, an online news headline company may run language models for news parsing, text classification for fake news detection, and personalized recommendation system for advertisement display.

To train large datasets or large models, the ML workloads are commonly run using distributed ML frameworks, *e.g.*, TensorFlow [1], MXNet [2] and Caffe2 [3]. In a distributed ML job, the dataset is divided and trained by separate workers, which exchange calculated model parameters with each other (either directly or through parameter servers (PSs)) to derive the global parameters. The workers and PSs may well be distributed onto different physical servers, when they cannot be completely hosted on one server, or to maximize resource fragment utilization on servers [4].

It is a fundamental challenge faced by cluster operators how to efficiently place different ML jobs onto servers to achieve high resource efficiency and training throughput. Many existing cluster schedulers (*e.g.*, Borg [5], Mesos [6]) tend to allocate more resources to the jobs than server resource capacity, in terms of resources such as CPU and memory, to maximize resource utilization (assuming not all jobs use their required resources fully at all time). However, even without over-subscription of resources, co-located ML jobs on the same server may interfere with each other negatively and experience performance unpredictability. This is because the jobs share underlying resources such as CPU caches, disk I/O, network I/O and buses (*e.g.*, QPI, PCIe), besides the resources typically considered by modern cluster schedulers. For example, when the GPU cards on a server are allocated to different ML jobs, the PCIe bus is shared when the jobs shuffle data between their allocated CPU and GPU; the QPI bus is shared when two allocated GPUs are not attached to the same CPU in the non-uniform memory access architecture.

Different levels of interference (*i.e.*, resource contention) occur when different types of ML jobs are co-located, depending on the models being trained and behavior of the training programs written by the users. Some ML jobs are CPU intensive, *e.g.*, CTC [7]; some are disk I/O intensive, *e.g.*, AlexNet [8], due to reading images for preprocessing; and some have a high network bandwidth consumption level, due to a large model size (number of parameters) and small mini-batch sizes (leading to more frequent parameter exchanges among workers), such as VGG-16 [9].

It is a natural idea to co-locate jobs with low levels of interference to optimize performance. However, existing schedulers used in practical ML clusters (*e.g.*, Yarn [10], Mesos [6]) are largely interference-oblivious, due mainly to the difficulty of obtaining potential interference levels of many jobs. In the literature, a number of work have showcased the potential and effectiveness of interference-aware scheduling, *e.g.*, considering network contention in MapReduce jobs [11] [12], cache access intensity of HPC jobs [13]. These studies build an explicit interference model of the target performance based on certain observations/assumptions and rely on hand-crafted heuristics for incorporating interference in scheduling [11] [13] [14]. They often require detailed application profiling under tens of interference sources, and careful optimization of coefficients in the performance model or thresholds in the heuristics accordingly. Generality is an issue with these white-box approaches: when the workload type or hardware configuration changes, the heuristics may not work well.

In this paper, we pursue a black-box approach for ML job placement that embraces interference while not relying on

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17204715, 17225516, C7036-15G (CRF), grants NSFC 61628209 and HKU URC Matching Funding.

detailed analytical performance modeling. Inspired by recent good results of deep reinforcement learning (DRL) in the game of Go [15] and video streaming [16], we adopt DRL in our scheduler design, and present *Harmony*, a deep learning-driven scheduler for ML clusters. *Harmony* encodes workload interference implicitly in a neural network (NN) that maps raw cluster and job states (*e.g.*, available resources, jobs' resource demands) to job placement decisions (in terms of which server to place each worker/PS of a job onto). Specifically, we make the following contributions in developing *Harmony*.

▷ We identify severe performance degradation when sharing resources among ML workloads, which has not been revealed in the existing literature. In contrast to previous heuristics that require operator insight and application knowledge, we propose a general design, *i.e.*, using DRL to schedule ML workloads, which can adapt to unknown dynamics (*e.g.*, interference not experienced before) automatically.

▷ We adopt a number of training techniques to resolve issues that may prevent DRL from converging to a good ML job placement policy, including actor-critic algorithm, job-aware action space exploration and experience replay. In view of a common lack of reward samples corresponding to different placement decisions, we build an auxiliary reward prediction model, which is trained using limited historical samples and used for producing reward for unseen placement.

▷ We have implemented a prototype of *Harmony* on Kubernetes [17] and evaluated *Harmony* on a GPU cluster, with real ML jobs running on MXNet [2], training models from different application domains. Our experiment results show that *Harmony* outperforms commonly adopted scheduling policies (*e.g.*, multi-resource bin packing) by 25% in terms of average job completion time under various settings.

II. Background and Motivation

A. Distributed Model Training

An ML job trains a model (*i.e.*, a deep NN) using a large amount of data to minimize a loss function iteratively. It is typically computation intensive and hence many ML frameworks have been designed for distributed training [1] [2] [3]. Most of them adopt PS architecture [4]: the training dataset is split among workers which train a local copy of the model parameters using allocated data, respectively (*i.e.*, data parallel training); the global model is partitioned to be updated by multiple PSs. At the beginning of each training iteration, each worker pulls the latest parameters from PSs to update its local copy; then it calculates gradients (parameter updates) over a small number of samples (*i.e.*, a mini-batch) and pushes the gradients to PSs. After receiving the gradients from all workers,[1] PSs use stochastic gradient descent (SGD) method (or its variants) to update global parameters. One mini-batch after another, a worker trains its data and pulls/pushes parameters/gradients from/to the PSs. After the entire training dataset has been processed once, one training epoch is done. The dataset is trained for multiple epochs until the model converges or a preset maximum number of epochs is reached.

[1]We focus on synchronous training, which are more widely deployed in real-world ML clusters, due to higher stability and model accuracy that synchronous training can achieve as compared to asynchronous training [18].

TABLE I: DL Jobs

Model	Application domain	Dataset
ResNet-50	image classification	ImageNet
VGG-16	image classification	ImageNet
ResNeXt-110	image classification	CIFAR10
Seq2Seq	machine translation	WMT17
CTC	sentence classification	mr
WLM	language modeling	PTB

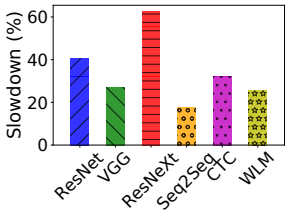


Fig. 1: Performance degradation: bin packing vs. standalone

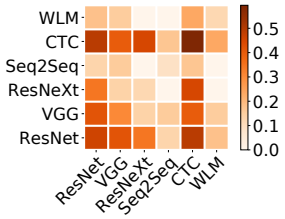


Fig. 2: Pair-wise interference (darker color indicates more severe interference)

B. Interference among Co-located ML Jobs

We showcase the impact of interference among co-located ML jobs with real ML workloads on our testbed (see Sec. VI for hardware details).

Case study 1: bin packing vs. standalone execution. We run 6 deep learning (DL) jobs, training ML models from official MXNet tutorials [19], as shown in Table I. Each job uses 1 PS and 1 worker for simplicity. In experiment 1, each job is run on a dedicated server; in experiment 2, the 6 jobs are packed onto 3 servers using multi-resource bin packing (*i.e.*, consolidating workloads on the least number of machines) [5] [20]. We compare each job's training speed in the two experiments, and show the slowdown percentage in Fig. 1, calculated as training speed (*i.e.*, number of trained epochs per unit time) in experiment 1 minus training speed in experiment 2, divided by the former. We observe a 33% performance degradation on average, and nearly 2x slowdown for training ResNeXt when the jobs are co-located. Note that when a job has multiple PSs and workers (instead of 1 PS and 1 worker in this case), the performance degradation would be more severe due to global synchronization, *i.e.*, bad placement of one worker/PS slows the overall training speed of a synchronous training job.

Case study 2: pair-wise interference level. We explicitly co-locate each pair of jobs training two different models in Table I on one server, and investigate the interference level, computed as the sum of the slowdown percentages of the two jobs (as compared to their respective standalone execution on the server). Fig. 2 shows that different levels of interference occur when different jobs are co-located, *e.g.*, ResNet and WLM are less affected when trained together, so do ResNet and Seq2Seq. This demonstrates good opportunities for optimizing resource efficiency and training performance in ML clusters by co-locating jobs with little interference.

Fig. 3: Placement under different schemes (diamonds represent PS and worker in job 1; squares represent those of job 2)

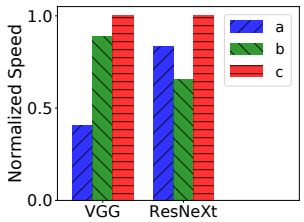


Fig. 4: Normalized training speed under 3 schemes

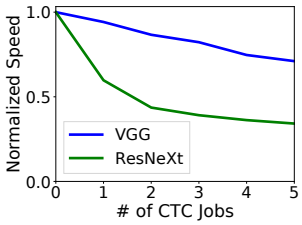


Fig. 5: Speed with increasing # of co-located CTC jobs

Case study 3: placement under representative policies. We further compare the results of three representative job placement policies (a) load balancing, *i.e.*, spreading workloads across servers as even as possible, as adopted in Mesos [6] and Kubernetes [17]; (b) multi-resource bin packing, as adopted in Google Borg [5] and Tetris [20]; (c) standalone execution. We run 2 ML jobs, each with 1 PS and 1 worker, on 2 servers; each server can accommodate up to 4 tasks (either PS or worker). The first job trains CTC and the second trains VGG-16 or ResNeXt-110.

Fig. 3 illustrates 3 placement schemes according to the policies. Fig. 4 shows the training speed when the second job runs VGG-16 and ResNeXt-110, respectively, under the 3 placement schemes. Standalone execution leads to the best performance, but also resource underutilization. Ideally, bin packing should outperform load balancing, in terms of both resource utilization and training performance, since it avoids cross-machine communication by placing PS and worker together. But this is not true when the second job trains ResNeXt-110, which performs better under the load balancing scheme. This is due to the more severe interference between training ResNeXt and CTC together. Fig. 5 further shows the training speed of ResNeXt (or VGG) when the number of co-located CTC training jobs increases. The more jobs are co-located, the worse the interference is.

All 3 schemes are not good enough to achieve high resource utilization and training speed at the same time. When more jobs are colocated (the number of different combinations of jobs would be huge), performance interference is even harder to identify and model. We resort to a black-box policy learned through DRL for job placement.

C. Challenges in Applying DRL in Job Placement

RL has been widely used for sequential decision making in an unknown environment, where an agent observes the current environmental state, selects an action based on current policy, and updates the policy based on the feedback (*i.e.*, reward from the environment). In DRL, the policy is learned through training an NN using rewards collected by trial-and-error interactions with the environment, with the goal of optimizing cumulative reward over time [21].

In our placement problem, the state space and action space are very large. For example, the action space is exponential in the numbers of jobs, workers/PSs in jobs, and servers. Even with 6 jobs, 3 workers plus PSs each, and 6 servers, there are more than 100 million different ways of placement. The complexity of state space and action space often prevents (quick) convergence of RL to a good decision making policy [22], due to insufficient or ineffective exploration. Further, it leads to significant practical difficulty in collecting enough traces for DRL training, which contain reward samples corresponding to various deployment ways. Even in production clusters that have operated for a few years, hardly ever all possible placement scenarios have happened. Without sufficient samples, it is unlikely to train the DRL NN to converge to a good policy.

To train our DRL model, we need one way to produce synthetic reward samples for placement decisions that are produced by DRL agent, but not seen in historical ML cluster operation traces. We do not rely on analytical interference models [11] [12], but adopt a more generic approach of using another NN for reward modeling, trained through supervised learning using the available traces.

III. System Overview

We consider a machine learning cluster with multiple GPU servers. ML training jobs are submitted to the cluster over time. Each job runs a distributed ML framework (*e.g.*, MXNet, as in our experiments) to learn a specific ML model by repeatedly training its dataset. We focus on distributed ML jobs using the parameter server architecture; our design can be readily extended to handle jobs using all-reduce type of algorithms for direct parameter exchange among the workers [3].

When submitting an ML job, the job owner provides the following job information: (i) his specific resource demands to run each worker and each PS, respectively, (ii) the total numbers of workers and PSs to use, and (iii) the total number of training epochs to train the dataset for. Worker/PS resource demands and total worker/PS numbers can be provided based on the model being trained, training program the user wrote, and experience of the user in training similar models. The total number of training epochs can be set based on expert knowledge or job history, *e.g.*, as an upper bound on the number of epochs used to achieve model convergence (typically indicated by the convergence of model loss or accuracy according to a threshold), when similar models were trained before.

Our proposed ML cluster scheduler, *Harmony*, schedules jobs in a batch processing manner, similar to [23] [18]. Time is divided into small *scheduling intervals*. *Harmony* batches newly arrived jobs in each interval and then decides the batch's placement altogether, *i.e.*, on which server each worker and each PS in a job should be run (as a virtual machine or a container). Then the jobs are deployed accordingly and run to completion, *i.e.*, placement of each job's workers and

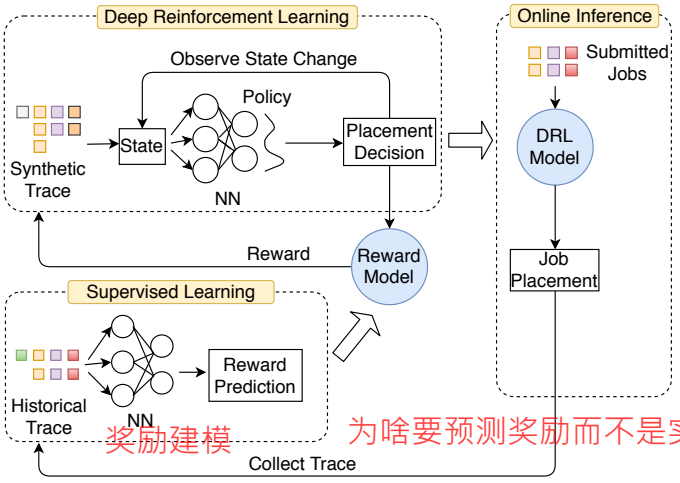


Fig. 6: *Harmony* workflow

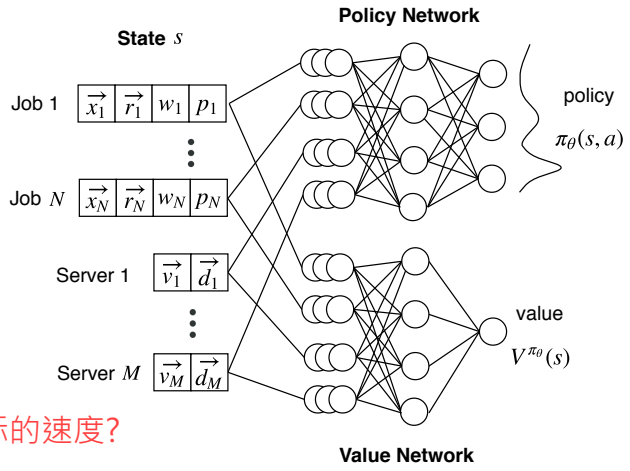


Fig. 7: DRL architecture

PSs does not change through the training course. Hence *Harmony* schedules newly arrived jobs only in each scheduling interval, placing them according to current resource availability on the servers. According to our discussions with companies operating large AI clouds (*e.g.*, Microsoft and Alibaba), job submission with resource specification and no placement adjustment after deployment are the norm in ML clusters; any dynamic adjustment of resource allocation to a running job is hard to implement in practice.

Harmony aims to minimize the average job completion time in the ML cluster, respecting the server capacity constraints. It learns a good job placement policy based on DRL, and combines offline training with online inference plus model update. The detailed workflow of *Harmony* is shown in Fig. 6.

Offline training is largely indispensable for producing a good model for online decision making (by inferences on the model); pure online learning of the policy NN from scratch has been widely known to result in poor policies at the beginning of learning, as DRL typically requires a large number of trials and errors in order to converge to a good policy [22] [15]. Large historical traces containing enough samples may not always be available; for DRL models with large action space, DRL may select actions or enter states that are never seen in practice. We boost our DRL with a reward prediction model (using another NN), to resolve the issue of insufficient training samples. Our offline training is divided into two steps.

▷ *Reward model training.* With historical job traces, *Harmony* trains the reward prediction NN using supervised learning. The input includes job set information and placement state; the label is the reward (training speed) of each job. This model provides fast reward evaluation for any job set with corresponding placement decisions.

▷ *DRL model training.* The DRL NN takes various job sets and cluster resource availability as input, and produces placement decisions for new jobs in the set. With the reward prediction model, we can effectively expand the available trace set and generate sufficient samples for DRL training.

Online inference and model update. The offline trained models are used for online decision making. In each schedule interval, *Harmony* decides placement of the new job batch via inference on the DRL NN, and observes actual rewards corresponding to the placement decisions. We periodically retrain the DRL NN and the reward NN using online collected samples, to continuously improve decisions over time.

We detail our design of DRL and reward prediction modules in the following sections.

IV. Deep Reinforcement Learning based Placement Policy

We first present our DRL algorithm for learning the job placement policy that maximizes job training speed across the entire cluster.

A. DRL Framework

Fig. 7 shows the detailed design of our DRL framework.

State space. The input state to the DRL NN is $s = (\boldsymbol{x}, \boldsymbol{r}, \vec{w}, \vec{p}, \boldsymbol{v}, \boldsymbol{d})$, includes the following:

(1) $\boldsymbol{x}$, an $N \times L$ binary matrix encoding the ML models trained by the jobs, where N is the maximal number of concurrent jobs in an interval and L is the maximal number of models that can be trained in the cluster (*i.e.*, total number of types of training jobs at all times). The concurrent jobs include both newly arrived jobs and uncompleted jobs which were submitted earlier; the reason to include existing jobs whose placement has been decided in earlier intervals, is to allow the DRL model to learn potential interference between new jobs and existing jobs on shared servers. Each vector $\vec{x}_n$ in $\boldsymbol{x}$, $\forall n = 1, \ldots, N$, is a one-hot encoding of job n's model [22]. The same ML model, *e.g.*, a DNN of the same architecture and mini-batch size but possibly different learning rates, uses the same encoding. For example, if there are 3 models in total and 3 concurrent jobs using each of the models respectively, then $\boldsymbol{x} = \{[1,0,0],[0,1,0],[0,0,1]\}$.

(2) $\boldsymbol{r}$, an $N \times 2(1+K)$ matrix encoding worker/PS resource demands in the jobs, where K is the number of resource types to compose a worker or a PS. In each vector $\vec{r}_n$ in $\boldsymbol{r}$, the first value represents the number of workers requested by job

n, and the next K values represent demand for the K types of resource in each worker; similarly, the rest $1+K$ values represent the number of PSs requested by job n and each PS's resource composition. For example, considering a job with 3 workers and 2 PSs, each requiring two types of resources (GPUs and CPU cores), $\vec{r}_n = [3,1,2,2,0,1]$ represents that each worker in the job requires 1 GPU and 2 CPU cores, and each PS needs no GPU but 1 CPU core.

(3) $\vec{w}$ ($\vec{p}$), an N-dimensional vector, in which the nth item is the number of workers (PSs) allocated to the nth job.

(4) $\boldsymbol{v}$, an $M \times K$ matrix representing available amount of each type of resources on the servers, where M is the number of physical servers. Each vector $\vec{v}_m$, $\forall m = 1,\ldots,M$, represents available resources on server m.

(5) $\boldsymbol{d}$, an $M \times 2N$ matrix encoding the placement of workers and PSs of the concurrent jobs on the servers. In each vector $\vec{d}_m$ $(m = 1,\ldots,M)$, the number of workers and the number of PSs of job n $(n = 1,\ldots,N)$ placed on server m, is on the $2n-1^{\text{th}}$ and $2n^{\text{th}}$ position, respectively. Suppose sever m hosts 1 PS and 1 worker of job 2 and 1 PS and 2 workers of job 5 among 6 jobs; we have $\vec{d}_m = [0,0,1,1,0,0,0,0,2,1,0,0]$.

Action space. After receiving s, the DRL agent selects an action a based on a policy $\pi_{\boldsymbol{\theta}}(s,a)$, which is a probability distribution over the action space. The policy is produced by an NN, and $\boldsymbol{\theta}$ is the set of parameters in the NN. Naturally, an action can include placement decisions of all new jobs in a scheduling interval (recall we do not adjust placement of existing jobs); however, this leads to an action space of exponential size, due to the exponentially many placement combinations of all workers and PSs in all jobs. A large action space may incur significant training time and worse results [22]. To expedite policy learning, we simplify the action definition, and our action space contains $2MN'$ actions as follows (N' denotes the maximal number of newly arrived jobs in an interval and $N' \leq N$): (i) $(n,0,m)$, meaning placing one worker of job n on server m, $\forall n \in [1,N'], m \in [1,M]$; (ii) $(n,1,m)$, placing one PS of job n on server m, $\forall n \in [1,N'], m \in [1,M]$.

In each scheduling interval, we allow multiple inferences over the NN, each selecting an action out of the action space, in order to come up with a complete set of placement decisions for all workers and PSs in the new jobs (or the inferences stop when there are not enough resources to place any additional PS/worker). In this way, we use multiple inferences to effectively reduce action space [23] [22].

Reward. We target average job completion time minimization by training the policy NN. Job completion time would be a natural reward to observe, but it is only known when a job is finished, which may well be hundreds of scheduling intervals later; further, completion time of a job is decided not only by the current job placement state, but also future deployment of upcoming jobs (possibly on the same servers and interfering with this job). We design a per-interval reward to collect more reward samples through the job processes, for more frequent RL model updates to expedite convergence. The reward r observed when action a is taken under state s is the sum of normalized training speeds of all concurrent jobs in the scheduling interval:

$$r = \sum_{n \in [N]} \frac{c_n}{e_n} \tag{1}$$

where c_n is the training speed (*i.e.*, number of trained epochs in this interval) of job n and e_n is the total number of training epochs that job n should complete. The rationale behind this reward design is that the more epochs a job trains in an interval, the fewer intervals it takes to complete, and hence maximizing cumulative reward amounts to minimizing average job completion time.

NN model. We design the policy NN architecture as follows (Fig. 7). The state of each job or each server is connected to a fully connected layer separately, and then they are connected to a few fully connected layers before the output layer. In this way, the NN can extract features from each job or each server before merged together as a table. To respect server resource capacities, in the output layer of the NN, we mask the invalid actions, which deploy a worker or PS on a server without enough resources to run it, by setting its probability to 0 in the policy distribution. Then we rescale the probabilities on all actions such that the sum still equals 1.

B. DRL Model Training

We apply the REINFORCE algorithm [24] to train the policy NN, which updates the NN parameters $\boldsymbol{\theta}$ using policy gradients computed with samples. Each sample is a four-tuple, (s,a,r,s'), where s' is the new state after action a is taken in state s. Note that our system runs differently from standard RL: we do multiple inferences (*i.e.*, produce multiple actions) using the NN in each scheduling interval t; the input state changes after each inference; we only observe the reward and update the NN once after all inferences in the interval are done. Let I_t be the set of inferences in interval t; then we can obtain I_t samples in the interval, and we set the reward in each of these samples to be the reward in (1) observed after all inferences are done in t.

The goal of training the policy NN is to maximize the expected cumulative discounted reward $J(\boldsymbol{\theta}) = \mathbb{E}[\sum_{i \geq 0} \gamma^i r_i]$, where $\gamma \in [0,1]$ is the discount factor, i indicates the total number of inferences done from system start, and r_i is the reward in the sample corresponding to the ith inference. The policy gradient of $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$, to be used for NN update in interval t, can be calculated as follows [24]:

$$\bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\sum_{i \in I_t} \bigtriangledown_{\boldsymbol{\theta}} \log(\pi_{\boldsymbol{\theta}}(s_i,a_i)) Q^{\pi_{\boldsymbol{\theta}}}(s_i,a_i)] \tag{2}$$

where the Q value, $Q^{\pi_{\boldsymbol{\theta}}}(s_i,a_i)$, represents the "quality" of the action a_i taken in given state s_i following the policy $\pi_{\boldsymbol{\theta}}$, calculated as the expected cumulative discounted reward to obtain after selecting action a_i at state s_i following $\pi_{\boldsymbol{\theta}}$, *i.e.*, $Q^{\pi_{\boldsymbol{\theta}}}(s_i,a_i) = \sum_{i' \geq i} \gamma^{i'-i} r_{i'}$. Since we can not enumerate all possible future states to calculate an exact Q value, we can use a mini-batch of samples to calculate an empirical Q

这个代码是怎么写的?

value [24] and then compute the gradient $\bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. We can then apply the SGD method to update parameters $\boldsymbol{\theta}$ to improve the empirical cumulative discounted reward. The idea behind is to increase the probabilities of selecting actions whose Q values are positive and reduce the probabilities of actions with negative Q values.

Beyond the basic policy gradient-based training, we adopt a number of techniques to stabilize training, expedite policy convergence, and improve the quality of the obtained policy.

1) Actor-critic: The REINFORCE algorithm may suffer from high variance in the Q values derived (for computing gradients), preventing quick convergence of the policy model [25]. To reduce the variance, we improve the basic policy gradient training with the actor-critic algorithm [21]. The basic idea is to introduce a baseline function dependent on the state, to improve the gradients used in SGD for updating the policy NN. We reinforce an action from a state if the "quality" of this action, $Q^{\pi_{\boldsymbol{\theta}}}(s_i, a_i)$, is better than the "average quality" of all possible actions in s_i, $V^{\pi_{\boldsymbol{\theta}}}(s_i)$, calculated as the expected cumulative reward following the policy π from state s_i, over all possible actions in the state. That is, we evaluate how good an action is by its advantage, *i.e.*, $Q^{\pi_{\boldsymbol{\theta}}}(s_i, a_i) - V^{\pi_{\boldsymbol{\theta}}}(s_i)$. We use this advantage instead of $Q^{\pi_{\boldsymbol{\theta}}}(s_i, a_t)$ in Eqn. (2) for gradient calculation. The purpose is to ensure a much lower variance in the estimation of the policy gradient, such that policy learning is more stable.

The value function, $V^{\pi_{\boldsymbol{\theta}}}(s_i)$, is typically estimated by a value network. It has the same architecture as the policy network, except that the output layer is a linear neuron without any activation function [25]. The input to the value network is the same as that to the policy network (see Fig. 7); the output of the value network is an estimate of value $V^{\pi_{\boldsymbol{\theta}}}(s_i)$ (while the output of the policy network is the policy distribution over placement actions). The value network is trained using temporal difference method [25]. TDerror

2) Exploration: To obtain a good policy through DRL, we need to ensure that the action space is adequately explored (*i.e.*, actions leading to high rewards can be sufficiently produced); as otherwise, DRL may well converge to poor local optimal policy [22] [25]. One approach to encourage exploration is to add an entropy regularization term $\beta \cdot \bigtriangledown_{\boldsymbol{\theta}} H(\pi_{\boldsymbol{\theta}}(s_i, \cdot))$ [25] into gradient calculation in Eqn. (2), where β is the entropy weight. The basic idea of entropy regularization is to encourage uniform action probability and prevents convergence to a single choice of output.

However, we find that entropy regularization is not enough due to the large exploration space for job placement. We further adopt another technique based on the ϵ-greedy method [21]. When performing each inference over the policy NN (for its training), with probability $1 - \epsilon$, we adopt the action (worker/PS placement decision) produced according to the NN's output policy distribution; with probability ϵ, we randomly choose between multi-resource bin packing and load balancing policies, and adopt the placement decision produced by the chosen policy. The bin packing policy places a worker or a PS (specified in the action selected by the NN) on a machine with least capacity left (where the worker or PS can still be accommodated); load balancing places one worker or PS on the least loaded machine. The rationale behind is to enable NN to effectively explore the tradeoff between resource utilization (bin packing is best for) and workload interference (that load balancing avoids). In this way, we improve exploration quality to guide the NN training to converge to a good policy.

3) Experience replay: It has been known that training an RL model using consecutive samples is hard to converge, due to the correlation among the samples [26]. The current policy NN determines the following training samples, *e.g.*, if the policy network finds that packing jobs improves reward, then the next sample sequence will be dominated by those produced from this strategy; this may lead to a bad feedback loop, preventing exploration of samples with higher rewards. We adopt experience replay [26] to alleviate correlation in the sample sequence.

Specifically, we maintain an FIFO replay buffer with a fixed size (*e.g.*, 8192 as in our experiments), large enough to buffer samples from multiple scheduling intervals. When computing the gradients for policy NN update in each scheduling interval, instead of using all samples collected in this interval, we randomly select a mini-batch of samples (32 samples as in our experiments) from the replay buffer, where the samples could be from multiple previous intervals.

V. Reward Prediction Model

We next design a reward model that can predict the reward given job and cluster states, based on which we can produce a large number of samples for DRL training. We adopt an NN as the reward model. An advantage of NNs is that they do not need hand-crafted features and can be applied directly to "raw" observations; besides, they are more general and can potentially be applied to other workloads.

NN architecture. The input state to the NN is a subset of the input to the DRL NN: $(\boldsymbol{x}, \vec{w}, \vec{p}, \boldsymbol{d})$. The resource demands of a worker and a PS in the concurrent jobs are not included as they can typically be inferred from a job's model type. The output is a vector including predicted training speeds of the input jobs (*i.e.*, number of training epochs to complete in an interval) The input state is connected to a sequence of hidden fully connected layers before the output layer. In practice, we find that fully connected layers work quite well in our scenario compared to more complicated neural layers, such as convolution layer (typically used for image processing [27]).

NN training. We train the NN by supervised learning using available samples in historical traces. We compare the predicted training speed c_n of each job n produced by the NN with the label c'_n, *i.e.*, training speed of each job n in the traces, by computing the relative error of the prediction and the label: $L(\boldsymbol{c}, \boldsymbol{c'}) = \frac{1}{|N|} \sum_{n \in [N]} \frac{|c'_n - c_n|}{c'_n}$. Then we use SGD to update parameters in the NN to minimize the overall relative error. We train the NN iteratively using the samples from the historical traces such that the prediction produced by the NN converges with an acceptable relative error (*e.g.*, 10%).

VI. Performance Evaluation

We evaluate *Harmony* using testbed experiments.

A. Implementation

Scheduler on Kubernetes. We implement *Harmony* using python on Kubernetes 1.7 [17] as a custom scheduler. We run workers and PSs on docker containers. Training datasets of jobs are stored in HDFS 2.8 [28]. At the beginning of a scheduling interval, *Harmony* queries unscheduled jobs and existing cluster state by sending HTTP requests to the Kubernetes API server and makes placement decisions using the trained policy network. The Kubernetes agents start the workers/PSs of each job on servers accordingly. *Harmony* updates the DRL and reward models using online collected data. We run each training job using the MXNet framework [2].

DRL Training. For offline training, we implement the DRL NN using libraries provided on TensorFlow [1]. The NN has 3 hidden layers with 196 neurons in the first two hidden layers and 128 neurons in the last hidden layer. We do parallel training of the DRL NN: we use 20 workers to generate samples (using the reward prediction model) and calculate gradients locally; the gradients are aggregated synchronously to obtain the global parameters. We adopt Adam optimizer [29] with a fixed learning rate of 0.0001, mini-batch size of 32 samples per worker, reward discount factor $\gamma = 0.9$, and an experience replay buffer size of 8192 samples. The greedy exploration factor ϵ and entropy weight β are set to 0.5 at the beginning and are annealed linearly during training.

Reward Model Training. We implement the reward NN using TensorFlow too. The reward NN has 3 hidden layers with 196 or 128 neurons in each hidden layer. We train it using Adam optimizer [29] with a fixed learning rate of 0.005 and a batch size of 32 samples. We also apply batch normalization for reward NN to accelerate convergence [27]. The traces for the reward model are collected on our GPU cluster: we generate jobs with random resource configurations, place them randomly and measure the training speed of each job; the number of jobs generated and their worker/PS configurations are sufficient to saturate resource capacity of our cluster.

B. Evaluation Methodology

Testbed. We build a testbed of 6 GPU servers connected by a Dell Networking Z9100-ON 100GbE switch. Each server has one 8-core Intel E5-1660 CPU, two GTX 1080Ti GPUs, 48GB RAM, one MCX413A-GCAT 50GbE NIC, one 480GB SSD and one 4TB HDD.

Workloads. By default, the jobs are submitted to the cluster in a uniform random manner with 3 jobs per interval on average. Each interval is 20 minutes long. Upon an arrival event, we randomly select a job from Table I and set its required number of workers and PSs randomly in $[1, 3]$ to generate a job variant. For jobs training large datasets (*e.g.*, ImageNet [30]), we downscale the datasets so that the training can be finished in a reasonable amount of time.

Baselines. We compare *Harmony* with the following policies:

– Load Balancing (LB): it assigns a worker/PS to the server with the least load. We normalize the usage of resources (*e.g.*, CPU, GPU) and sum them as the load of a server.

– Tetris [20]: it uses multi-resource bin packing to place a worker/PS to avoid resource fragmentation.

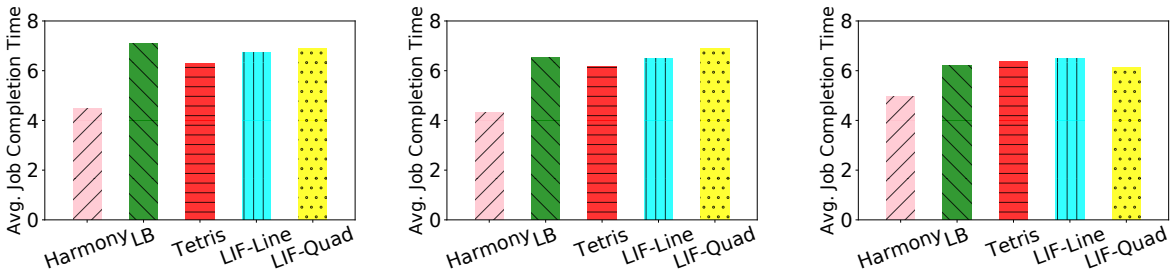
– Least Interference First (LIF-Line, LIF-Quad) [31] [11]: it builds a linear or non-linear interference model by assuming that task slowdown is a function of CPU and bandwidth usage. To determine coefficients in the interference model, we profile the performance of each ML model under different CPU and bandwidth usages and use a least-square solver to calculate the coefficients based on profiling data. We find that linear function and quadratic function fit our measured traces best, so we use them as two baselines, *i.e.*, LIF-Line and LIF-Quad. When placing a worker/PS, the algorithm calculates an interference score (*i.e.*, performance slowdown of all workers/PSs on a server) for each server and selects the server with the least interference.

C. Performance

Fig. 8 compares the performance of *Harmony* with baselines under three job arrival patterns: (1) default uniform distribution; (2) a Poisson process with an arrival rate of 2 per scheduling interval; (3) the job arrival process extracted from Google cluster traces [32], with downscaled arrival rates. *Harmony* improves average job completion time (in terms of number of intervals) by more than 25% compared with all baselines in the three cases. *Harmony* finds a good balance between load balancing and bin packing via exploration, to jointly reduce the computation interference and data transmission time, and improves its scheduling policy by feedback obtained after each decision making. Load balancing performs worse than *Harmony* as it only focuses on reducing computation interference on each machine, without considering network transmission overhead. Tetris tries to put the jobs together to avoid wasting resource fragments, and ignores performance degradation caused by interference. LIF-Line and LIF-Quad rely heavily on accurate modeling of interferences among jobs.

We also compare the reward model with interference model in LIF-Line and LIF-Quad. We shuffle the collected trace (13177 samples) and split it into training dataset (90%) and test dataset (10%). Fig. 9(a) shows that our model achieves 9.8% relative error, much lower than that of the interference models in LIF-Line and LIF-Quad. Further, Fig. 9(b) shows the relative errors incurred for predicting speeds of jobs training two models, Seq2Seq and CTC. We see that interference models in LIF-Line and LIF-Quad work well for CTC, but have large error on Seq2Seq, due to the following: CTC is CPU-intensive while Seq2Seq does not utilize CPU much and carries out computation mostly on GPU; LIF builds interference models for CPU and network sharing only.

In addition, to validate scalability of *Harmony*, we generate simulated traces of job placement on 30 machines (at a scale of a rack) based on analytical models [31] [11], and use the simulated traces to train reward prediction NN and our DRL NN. We observe that *Harmony* can still reduce average job



(a) Uniform (b) Poisson (c) Google trace

Fig. 8: Performance comparison under different job arrival patterns

completion time by at least 23%, as compared to the baselines. We omit detailed figures due to space limit.

D. Deep Dive

We next evaluate our detailed design of *Harmony*.

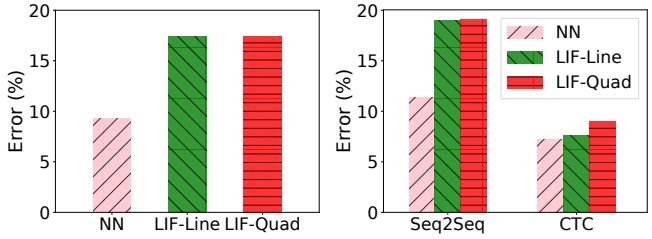
Number of neurons. We fix the number of hidden layers to 3 and vary the number of neurons in the hidden layer. We train all neural networks until convergence over the same training set. Fig. 10(a) shows the best performance is achieved when there are 128 neurons. With fewer neurons, there are not enough NN parameters to approximate the placement policy. The performance degrades with too many neurons, as it may capture unnecessary features (*i.e.*, overfitting).

Number of hidden layers. We fix the number of neurons to 196 in the first two hidden layers and 128 in the remaining hidden layers, and vary the number of hidden layers. As shown in Fig. 10(b), the NN with 3 hidden layers performs the best. With fewer NN layers, there are not enough parameters to approximate a good policy; with more NN layers, it generally takes a longer time to converge and results in lower performance due to overfitting.

Value network. To investigate how the value network affects training, we do not train the value network to provide the baseline, but use the exponential moving average of the rewards as a baseline in computing the gradient when training the policy network. In Table II, the first row shows average job completion time with all training techniques applied. We see that without the value network, the performance is 49.3% worse. This is because the average reward is not always an effective baseline; in some cases even the optimal action leads to a lower reward than the average reward over the history.

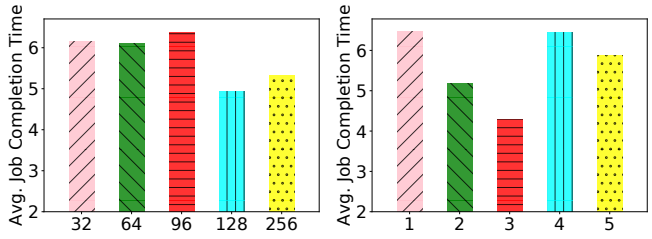
Exploration. We examine how exploration contributes to the performance. From Table II, we see that without exploration, the performance is significantly worse (*i.e.*, 68.2% slowdown). The reason is that without exploration, the DRL NN may make a lot of useless trials to try many obviously bad actions and get easily stuck in a local optimal policy.

Experience replay. We disable experience replay to examine its effectiveness in DRL training. As shown in Table II, without experience replay, the average job completion time is increased by 37.7%, indicating that disrupting the order of samples and using samples among different scheduling intervals to update NN is critical for our DRL training.



(a) Overall error (b) Errors for jobs training Seq2Seq and CTC

Fig. 9: Training speed prediction



(a) # of neurons (b) # of hidden layers

Fig. 10: Deep dive

TABLE II: Effectiveness of Training Techniques

Without	Avg. Job Completion Time (intervals)	Slowdown (%)
-	4.3	0
Value network	6.4	49.3
Exploration	7.2	68.2
Experience replay	5.9	37.7

VII. Related Work

ML job scheduling. There have been a few recent work on resource allocation in ML clusters. Dorm [33] is a utilization-fairness optimizer to schedule ML jobs. OASiS [34] is an online scheduling algorithm for ML jobs. SLAQ [35] and Optimus [18] build a performance model for each ML job and dynamically adjusts resource allocation. They focus on dynamic adjustment of the numbers of workers/PSs to fully utilize cluster resources or improve training quality. Instead, we study interference between colocated ML jobs and optimize job placement using DRL, given fixed resource demands.

Interference-aware task placement. Abhishek et al. [13]

design an interference-aware VM placement algorithm for high performance computing workloads in clouds. They heuristically classify workloads into several categories (*e.g.*, based on CPU cache interference level) and place workloads with little CPU interference together. Bu et al. [11] target network interference and locality-aware scheduling for MapReduce workloads. Xu et al. [36] build an analytical model for MapReduce applications by considering resource utilization and VM interference. Paragon [14] and Quasar [37] use collaborative filtering to predict application performance. These studies model interference explicitly and typically require application profiling to determine coefficients in the models. Instead, we leverage historical data traces and learn interference implicitly in NN without loss of generality.

DRL. DRL has achieved promising results in different problem domains. Mao et al. [38] use DRL to set task parallelism level and execution order for data-parallel jobs running on Spark. Liu et al. [39] use DRL to design a dynamic power management policy for data centers. Mao et al. [16] apply DRL to adjust streaming rates to cope with unstable network bandwidth in an adaptive video streaming system. Mirhoseini et al. [40] use DRL to optimize the operator placement of a TensorFlow computation graph in a single machine. Xu et al. [26] apply DRL for routing path selection in traffic engineering. These work assume enough training data for DRL, typically generated by a simulation model or online measurement. Instead, we show that interference is difficult to model and we develop a reward prediction NN to generate samples for DRL training.

VIII. Conclusion

This paper presents *Harmony*, a deep learning-based scheduler that addresses performance interference and minimizes average job completion time by efficient job placement in an ML cluster. Instead of designing placement policy based on analytical models of workload interference, we design a two-step learning mechanism: we first exploit limited historical traces to learn a reward neural network using supervised learning; then we train the DRL model to learn placement decisions using reward samples provided by the reward model. We believe that our reward prediction module design is general, and applicable to other DRL problems where historical traces are not sufficient. Evaluation on a Kubernetes cluster shows that *Harmony* outperforms representative scheduling policies by 25%, in reducing average job completion time in the cluster.

References

[1] M. Abadi, P. Barham *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *Proc. of USENIX OSDI*, 2016.

[2] T. Chen, M. Li *et al.*, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," in *NIPS Workshop on Machine Learning Systems (LearningSys)*, 2016.

[3] "Caffe2," https://caffe2.ai/, 2018.

[4] M. Li, D. G. Andersen *et al.*, "Scaling Distributed Machine Learning with the Parameter Server," in *Proc. of USENIX OSDI*, 2014.

[5] A. Verma, L. Pedrosa, M. Korupolu *et al.*, "Large-Scale Cluster Management at Google with Borg," in *Proc. of ACM EuroSys*, 2015.

[6] B. Hindman, A. Konwinski *et al.*, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *Proc. of USENIX NSDI*, 2011.

[7] Y. Kim, "Convolutional Neural Networks for Sentence Classification," in *Proc. of SIGDAT EMNLP*, 2014.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proc. of NIPS*, 2012.

[9] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-scale Image Recognition," in *Proc. of ICLR*, 2015.

[10] V. K. Vavilapalli, A. C. Murthy, C. Douglas *et al.*, "Apache Hadoop Yarn: Yet Another Resource Negotiator," in *Proc. of ACM SoCC*, 2013.

[11] X. Bu, J. Rao, and C.-z. Xu, "Interference and Locality-aware Task Scheduling for MapReduce Applications in Virtual Clusters," in *Proc. of ACM HPDC*, 2013.

[12] F. Xu, F. Liu *et al.*, "Network-Aware Task Assignment for MapReduce Applications in Shared Clusters," *Journal of Internet Technology*, 2015.

[13] A. Gupta, L. V. Kale, D. Milojicic, P. Faraboschi *et al.*, "HPC-aware VM Placement in Infrastructure Clouds," in *Proc of. IEEE IC2E*, 2013.

[14] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters," *ACM SIGPLAN Notices*, 2013.

[15] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang *et al.*, "Mastering the Game of Go Without Human Knowledge," *Nature*, 2017.

[16] H. Mao, R. Netravali, and M. Alizadeh, "Neural Adaptive Video Streaming with Pensieve," in *Proc. of ACM SIGCOMM*, 2017.

[17] "Kubernetes," https://kubernetes.io, 2018.

[18] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an Efficient Dynamic Resource Scheduler for Deep Learning Clusters," in *Proc. of ACM EuroSys*, 2018.

[19] "MXNet Official Examples," https://github.com/apache/incubator-mxnet/tree/master/example, 2017.

[20] R. Grandl, G. Ananthanarayanan, S. Kandula *et al.*, "Multi-Resource Packing for Cluster Schedulers," in *Proc. of ACM SIGCOMM*, 2014.

[21] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press Cambridge, 1998.

[22] O. Vinyals, T. Ewalds, S. Bartunov *et al.*, "StarCraft II: A New Challenge for Reinforcement Learning," *arXiv preprint arXiv:1708.04782*, 2017.

[23] H. Mao, M. Alizadeh, I. Menache *et al.*, "Resource Management with Deep Reinforcement Learning," in *Proc. of ACM HotNets*, 2016.

[24] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy Gradient Methods for Reinforcement Learning with Function Approximation," in *Proc. of NIPS*, 2000.

[25] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," in *Proc. of ICML*, 2016.

[26] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven Networking: A Deep Reinforcement Learning based Approach," in *Proc. of IEEE INFOCOM*, 2018.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proc. of IEEE CVPR*, 2016.

[28] "HDFS," https://wiki.apache.org/hadoop/HDFS, 2014.

[29] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *Proc. of ICLR*, 2015.

[30] "ImageNet Dataset," http://www.image-net.org, 2018.

[31] R. Chiang *et al.*, "TRACON: Interference-Aware Schedulingfor Data-Intensive Applicationsin Virtualized Environments," *IEEE TPDS*, 2014.

[32] C. Reiss, A. Tumanov *et al.*, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proc. of ACM SoCC*, 2012.

[33] P. Sun, Y. Wen, N. B. D. Ta, and S. Yan, "Towards Distributed Machine Learning in Shared Clusters: A Dynamically-Partitioned Approach," in *Proc. of IEEE Smart Computing*, 2017.

[34] Y. Bao, Y. Peng, C. Wu, and Z. Li, "Online Job Scheduling in Distributed Machine Learning Clusters," in *Proc. of IEEE INFOCOM*, 2018.

[35] H. Zhang, L. Stafman, A. Or *et al.*, "SLAQ: Quality-Driven Scheduling for Distributed Machine Learning," in *Proc. of SoCC*, 2017.

[36] F. Xu, F. Liu, and H. Jin, "Heterogeneity and Interference-Aware Virtual Machine Provisioning for Predictable Performance in the Cloud," *IEEE Transactions on Computers*, 2016.

[37] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-aware Cluster Management," *ACM SIGPLAN Notices*, 2014.

[38] H. Mao, M. Schwarzkopf, S. Venkatakrishnan *et al.*, "Learning Graph-based Cluster Scheduling Algorithms," in *Proc. of SysML*, 2018.

[39] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang *et al.*, "A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning," in *Proc. of IEEE ICDCS*, 2017.

[40] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner *et al.*, "Device Placement Optimization with Reinforcement Learning," in *Proc. of ICML*, 2017.