Deep Learning-Based Job Placement in Distributed Machine Learning Clusters With Heterogeneous Workloads

Yixin Bao, Yanghua Peng, and Chuan Wu, *Senior Member, IEEE*

***Abstract*—Nowadays, most leading IT companies host a variety of distributed machine learning (ML) workloads in ML clusters to support AI-driven services, such as speech recognition, machine translation, and image processing. While multiple jobs are executed concurrently in a shared cluster to improve resource utilization, interference among co-located ML jobs can lead to significant performance downgrade. Existing cluster schedulers, such as YARN and Mesos, are interference-agnostic in their job placement, leading to suboptimal resource efficiency and usage. Some literature has studied interference-aware job placement policy, but relies on detailed workload profiling and interference modeling, which is not a general solution. In this work, we present *Harmony*, a deep learning-driven ML cluster scheduler that places heterogeneous training jobs (either with parameter server architecture or all-reduce architecture) in a manner that minimizes interference and maximizes performance (*i.e.*, training completion time minimization). The design of *Harmony* is based on a carefully designed deep reinforcement learning (DRL) framework enhanced with reward modeling. The DRL integrates a dynamic sequence-to-sequence model with the state-of-the-art techniques to stabilize training and improve convergence, including actor-critic algorithm, job-aware action space exploration, multi-head attention, and experience replay. In view of a common lack of reward samples corresponding to different placement decisions, we build an auxiliary sequence-to-sequence reward prediction model, which is trained with 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 16%-42% in terms of average job completion time.**

***Index Terms*—Distributed machine learning systems, job scheduling, deep reinforcement learning.**

I. Introduction

IN LEADING IT companies, production machine learning (ML) clusters of GPU servers commonly host a variety of distributed ML workloads, to support the company's services. For example, an online video platform may use a speech recognition model to generate caption and run language models for auto-translation, image classification for illegal video detection, and personalized recommendation system for advertisement display.

Manuscript received 14 December 2020; revised 26 January 2022; accepted 29 July 2022; approved by IEEE/ACM Transactions on Networking Editor E. Kalyvianaki. Date of publication 8 September 2022; date of current version 18 April 2023. This work was supported in part by the Hong Kong Research Grants Council (RGC) under Contract HKU 17204619, Contract 17208920, and Contract 17207621. *(Corresponding author: Yixin Bao.)*

The authors are with the Department of Computer Science, The University of Hong Kong, Hong Kong (e-mail: yxbao@cs.hku.hk; yhpeng@cs.hku.hk; cwu@cs.hku.hk).

Digital Object Identifier 10.1109/TNET.2022.3202529

In order to accelerate training process over large datasets or large models, the ML workloads are commonly run using distributed ML frameworks, *e.g.*, TensorFlow [1], MXNet [2] and PyTorch [3]. In a distributed ML job, the dataset is divided and trained by separate workers, where the worker exchanges calculated model parameters with each other (either directly with an all-reduce collective or through parameter servers) to derive the global parameters. In case of a parameter server (PS) architecture, it is common that the workers and parameter servers may well be distributed onto different physical servers, either when they cannot be completely hosted on one server, or to maximize resource fragment utilization on servers [4], [5], [6].

With increasing deep learning (DL) training workloads hosted on public clouds or private clusters, a fundamental challenge faced by cluster operators is that, how to efficiently place different ML jobs onto servers to achieve high resource efficiency and training throughput. To maximize resource utilization, many existing cluster schedulers, such as Borg [7] and Mesos [8], tend to allocate more resources to the jobs than server resource capacity (*i.e.*, resource overselling), in terms of resources such as CPU and memory by assuming that 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 deterioration unpredictably. 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 (*e.g.*, GPU, memory). 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.

The levels of interference (*i.e.*, resource contention) are different when different types of ML jobs are co-located, due mainly to the models being trained and behavior of the training programs written by the users. Some ML jobs are CPU intensive, *e.g.*, CTC [9], due to demanding computation; some are disk I/O intensive, *e.g.*, AlexNet [10], due to reading images for pre-processing; 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 [11].

To optimize training performance, a natural idea is to co-locate jobs with low levels of interference. However, existing schedulers used in practical ML clusters (*e.g.*, Yarn [12], Mesos [8]) are largely interference-oblivious, due mainly to the difficulty of obtaining potential interference levels of many jobs, leading to longer job training time and sub-optimal resource efficiency. 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 [13], [14], cache access intensity of HPC jobs [15]. These studies build an explicit interference model of the target performance based on certain observations or assumptions (*e.g.*, that interference slows down performance exponentially [13]), and rely on hand-crafted heuristics for incorporating interference in scheduling [13], [15], [16]. To achieve high-quality scheduling decisions, such approaches often require detailed application profiling under tens of interference sources (*e.g.*, due to sharing of different resources such as CPU caches, network bandwidth), and careful optimization of coefficients in the performance model or thresholds in the heuristics accordingly. Generality is an issue with these white-box approaches: in case of the workload type or hardware configuration changes, the heuristics may not work well.

Different from existing approaches, in this paper, we pursue a black-box approach for ML job placement that embraces interference while not relying on detailed analytical performance modeling. Inspired by recent good results of deep reinforcement learning (DRL) in the game of Go [17], job scheduling [18], [19], [20], and video streaming [21], we adopt DRL in the design of our scheduler. Specifically, we 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 or parameter server of a job onto). *Harmony* utilizes DRL to learn to take near-optimal actions according to run-time information without relying on any mathematical model or any pre-defined job placement policy. 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 with a sequence-to-sequence based model to schedule ML workloads, which can adapt to unknown dynamics (*e.g.*, interference not experienced before) automatically.

▷ We propose a Transformer-based [22] DRL model to handle online job arrival sequence, without the need of knowing the number of arrived jobs beforehand. To extract the correlation among all concurrent jobs, we utilize the attention mechanism to obtain the interrelation of jobs at different positions of the online job sequence, by looking at the global information over the job sequence.

▷ We adopt a number of training techniques to facilitate DRL for converging to a good ML job placement policy, including actor-critic algorithm, job-aware action space exploration, multi-head attention and experience replay. In view of a common lack of reward samples corresponding to different placement decisions, we build an auxiliary sequence-to-sequence 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 [23] and evaluated *Harmony* on a GPU cluster, with real ML jobs running on MXNet [2], training models from different application domains (see Table I). Our experiment results show that *Harmony* outperforms commonly adopted scheduling policies (*e.g.*, multi-resource bin packing) by 16%-42% in terms of average job completion time under various settings.

II. Background and Related Work

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 parameter server architecture [4] or all-reduce collective [24]. In parameter server architecture, 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 parameter servers. At the beginning of each training iteration, each worker pulls the latest parameters from parameter servers 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 parameter servers. After receiving the gradients from all workers,[1] parameter servers 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 parameter servers. 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. While in all-reduce architecture, each worker maintains one copy of global model parameters and synchronize gradients in a peer-to-peer manner without parameter server. Specifically, each worker sums (or averages) the gradients from all other workers, and distributes the aggregated (or averaged) gradients to other workers. With such all-reduce operation, the target arrays of gradients in all workers are reduced to a single array of gradients, and the resultant gradients array are further returned to all workers. The most representative algorithm to implement this operation is ring all-reduce, which is most widely adopted in ML frameworks, including Horovod [6], PaddlePaddle [26], etc.

B. ML Job Scheduling

There have been a few recent work on resource allocation in ML clusters. Dorm [27] is a utilization-fairness optimizer to

[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 [25].

schedule ML jobs. OASiS [28] is an online scheduling algorithm for ML jobs, by scaling the number of workers and parameter servers to minimize job completion time. SLAQ [29] and Optimus [25] build a performance model for each ML job and dynamically adjusts resource allocation. They focus on dynamic adjustment of the numbers of workers/parameter servers to fully utilize cluster resources or improve training quality. Gandiva [30] proposes an introspective scheduling framework for deep learning jobs to continuously refine the scheduling decision. Tiresias [31] is an information-agnostic resource manager for GPU clusters. Or *et al.* [32] present an auto-scaling system for distributed deep learning to save resource costs and shorten job completion time. Themis [33] is a scheduler to ensure fairness in job completion time of ML workloads in a shared GPU cluster. Chaudhary *et al.* [34] propose a fair share scheduler to balance efficiency and fairness in GPU clusters for deep learning training. Gavel [35] is a heterogeneity-aware scheduler that systematically generalizes a wide range of existing scheduling policies, and is able to optimize for many high-level metrics like fairness, make span, and cost. Instead, we study interference between co-located ML jobs and optimize job placement using DRL, given fixed resource demands, which is not covered in the above schedulers.

C. Interference-Aware Task Placement

Previous work have showcased the potential of interference-aware scheduling. For example, Gupta *et al.* [15] 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. Mihailescu *et al.* [36] propose OX, a runtime system to mitigate network congestion while satisfying availability requirements for data analytics, by discovering communication topology of each application and migrating VMs between racks to optimize communication. Bu *et al.* [13] target network interference and locality-aware scheduling for MapReduce workloads. They assume interference affects application performance in a non-linear way and construct an interference model to predict slowdown. Xu *et al.* [37] build an analytical model for MapReduce applications by considering resource utilization and VM interference. Paragon [16] and Quasar [38] 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.

D. DRL

DRL has achieved promising results in different problem domains, including games [17], [39], resource allocation [18], device placement [40], video streaming [21], and traffic engineering [41]. Specifically, Mao *et al.* [18] use DRL to set task parallelism level and execution order for data-parallel jobs running on Spark. Liu *et al.* [42] use DRL to design a dynamic power management policy for data centers. Mao *et al.* [21] apply DRL to adjust streaming rates to cope with unstable network bandwidth in an adaptive video streaming system.

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

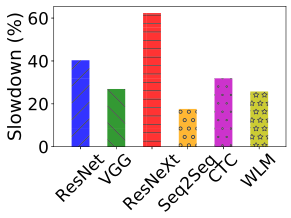


Fig. 1. Performance degradation: bin packing vs. standalone.

Mirhoseini *et al.* [40], [43] use DRL to optimize the operator placement of a TensorFlow computation graph in a single machine. Xu *et al.* [41] apply DRL for routing path selection in traffic engineering. Peng *et al.* [19] and Gong *et al.* [20] propose DRL-based online scheduling algorithms for deep learning jobs, where they only consider parameter server architecture and consider a fixed number of newly arrived jobs at each time. Wang *et al.* [44] propose a scheduler based on deep reinforcement learning to find a trade-off between model quality and cost in serverless computing. 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.

III. MOTIVATION

A. 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. VII for hardware details).

Case study 1: bin packing vs. standalone execution. We run 6 deep learning (DL) jobs with parameter server architecture, training ML models from official MXNet tutorials [45], as shown in Table I. Each job uses 1 parameter server 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) [7], [46]. 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 show the slowdown instead of absolute training completion time in order to normalize the different completion times of the jobs. 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 parameter servers and workers (instead of 1 parameter server and 1 worker in this case), the performance degradation would be more severe due to global synchronization, *i.e.*, bad placement of one worker/parameter server

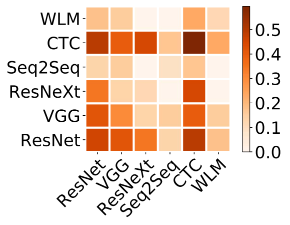


Fig. 2. Pair-wise interference (darker color indicates more severe interference).

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.*, ResNeXt and WLM are less affected when trained together, so do ResNeXt and Seq2Seq. This demonstrates good opportunities for optimizing resource efficiency and training performance in ML clusters by co-locating jobs with little interference. Besides, all jobs are severely affected by the CPU-intensive CTC job, indicating that a standalone execution for the CTC job may be better.

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 [8] and Kubernetes [23]; (b) multi-resource bin packing, as adopted in Google Borg [7] and Tetris [46]; (c) standalone execution. We run 2 ML jobs, each with 1 parameter server and 1 worker, on 2 servers; each server can accommodate up to 4 tasks (either parameter server or worker). The first job trains CTC and the second trains VGG-16 or ResNeXt-110. Note that each job's PS and worker may be placed on different servers depending on the placement policies.

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 parameter server and worker together. This is true when the first job trains VGG-16, which involves large amount of gradient exchanges in each training iteration and is sensitive to network latency and throughput. However, it 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 CPU interference when training ResNeXt and CTC together. Fig. 5 further shows the training speed of ResNeXt (or VGG-16) when the number of co-located CTC training jobs increases. We see that (1) the more jobs are co-located, the worse the interference is; (2) ResNeXt training is more sensitive to CPU-intensive workloads than VGG-16 training.

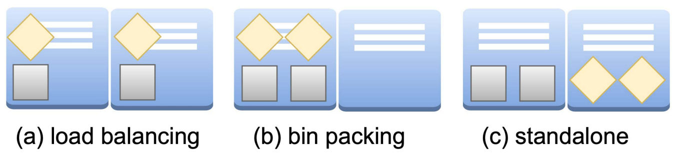


Fig. 3. Placement under different schemes (diamonds represent parameter server and worker in job 1; squares represent those of job 2).

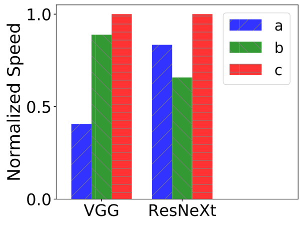


Fig. 4. Normalized training speed under 3 schemes.

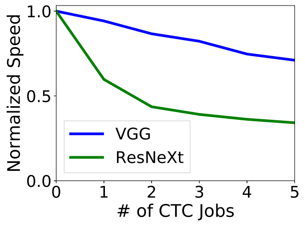


Fig. 5. Speed with increasing # of co-located CTC jobs.

All 3 schemes are not good enough to achieve high resource utilization and training speed at the same time. When more jobs are co-located (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.

B. 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 [47].

Harmony aims to handle online arrived jobs, which means that we do not assume knowledge of total number of jobs at different time. In this case, a straightforward solution is to train one NN to produce placement decisions for each newly arrived job independently. However, this approach may not work well since it ignores the interference among different jobs (as what we focus on). It is quite challenging to use DRL to dynamically process multiple jobs which may come and go at any time, since the agent usually uses a deep feed-forward NN or a convolutional NN to produce an action, which requires a fixed input size. Using a recurrent neural network or long short-term memory [48] can deal with an input sequence with variable sequence length, but earlier jobs' information at the beginning of the input sequence becomes less important after we have iteratively compressed all information into a fixed-size matrix. A sequence with dozens of jobs represented by a small matrix will surely lead to information loss, inadequate representation, etc. Instead, we adopt the Transformer structure with the attention mechanism [22] to model the long-range dependencies among the new jobs without regard to their separation in the input job sequence. In this way, we can have a global view of all the concurrent jobs' information, available resources, and existing placement decisions.

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/parameter servers in jobs, and servers. Even with 6 jobs, 3 workers plus parameter servers 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 [39], 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. In this way, we can obtain the reward sample immediately instead of running a set of jobs in a real cluster. We do not rely on analytical interference models [13], [14], but adopt a more generic approach of using another NN for reward modeling, trained through supervised learning using the available traces. To expedite convergence of our DRL model to a good placement policy, we strategically design some learning techniques, that enable efficient exploration of the large action space and full exploitation of good feedback.

IV. System Overview

We consider a machine learning cluster with multiple GPU servers, where the ML training jobs are submitted into the cluster over time in an online fashion. 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 either the parameter server architecture or all-reduce architecture, and data parallel training; our design can be readily extended to handle jobs using model parallel training [5].

Along with a submitted ML job, the job owner provides the following job information: (i) his/her specific resource demands to run each worker and each parameter server, respectively, (ii) the total numbers of workers and parameter servers to use, and (iii) the total number of training epochs to train the dataset for. For example, a worker often requires at least 1 GPU, and a parameter server needs high bandwidth. The resource demands of worker/parameter server and total worker/parameter server 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 training 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

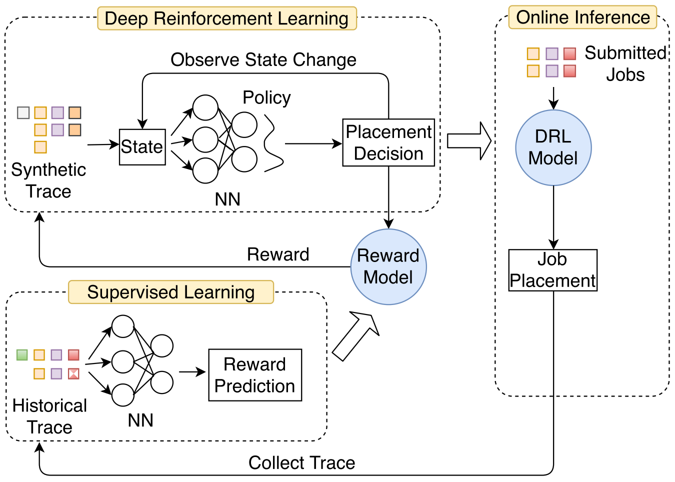


Fig. 6. *Harmony* workflow.

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 [25], [49], and [19]. Time is divided into small *scheduling intervals*. *Harmony* does not assume that job arrival time is known a prior, and batches newly arrived jobs in each interval. Then it decides the job placement of the whole batch altogether, *i.e.*, on which server each worker and each parameter server in a job should be run (as a virtual machine or a container). Then the jobs are deployed accordingly based on placement decisions, and *Harmony* runs these jobs to completion, *i.e.*, the placement of each job's workers and parameter servers 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 and existing placement of previously arrived jobs on the servers. According to our discussions with companies operating large AI clouds (*e.g.*, Microsoft [50] and Alibaba [51]), job submission with resource specification and no placement adjustment after deployment are the norm in ML clusters, mainly due to two reasons: (1) Stopping a training job and then resuming the training requires modifications of ML frameworks and user code in order to recover the correct training state (e.g., number of epochs and iterations that have trained, random seed, etc.). (2) Stopping and resuming a training job brings significant overhead, including rescheduling time, container restart, dataset reloading, checkpoint reloading, etc [52]. Dynamic resource adjustment without much overhead to a running job is hard to implement in practice.

In *Harmony*, we aim 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 model training, and combines offline training with online inference plus model update. The detailed workflow of *Harmony* is shown in Fig. 6.

Offline training is largely essential for producing a good model for online decision making (*i.e.*, by inferences on the model). Nevertheless, pure online learning of the policy NN from scratch, *i.e.*, RL with online data, 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 [17], [39]. For example, AlphaGo Zero uses 29 million games in offline training [17], and DeepRM uses 20 thousand samples for a simple NN with 1 hidden layer [49]. However, 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, while we still need to provide the reward for the state and selected action as feedback from the environment for further self-improvement. 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 the existing placement on different servers; the label is the reward (training speed) of each concurrent 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, existing placement, and cluster resource availability as input, and produces placement decisions for all new jobs in the set. We estimate reward based on the reward model, and obtain reward for DRL training. With the reward prediction model, we can effectively expand the available trace set and generate sufficient samples for DRL offline training.

Online inference and model update. The offline trained models are then used for online decision making. In each scheduling interval, *Harmony* decides placement of the new job batch via inference on the DRL NN, and observes actual rewards corresponding to the placement decisions. In this way, new reward samples are collected, which may not have been seen historically and hence not used for training in the offline phase. We periodically retrain the DRL NN and the reward NN using online collected samples, to continuously improve decisions over time.

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

V. 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 a sequence $s = (s_1, \ldots, s_N)$. The length of the sequence, N, is the number of concurrent jobs in the current time. 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 time, is to allow the DRL model to learn potential interference between new jobs and existing jobs on shared servers. $s_n = (\vec{x}_n, \vec{r}_n, w_n, p_n, \boldsymbol{v}, \vec{d}_n, u_n), \forall n \in [N]$ is the state of n^{th} job, includes the following:

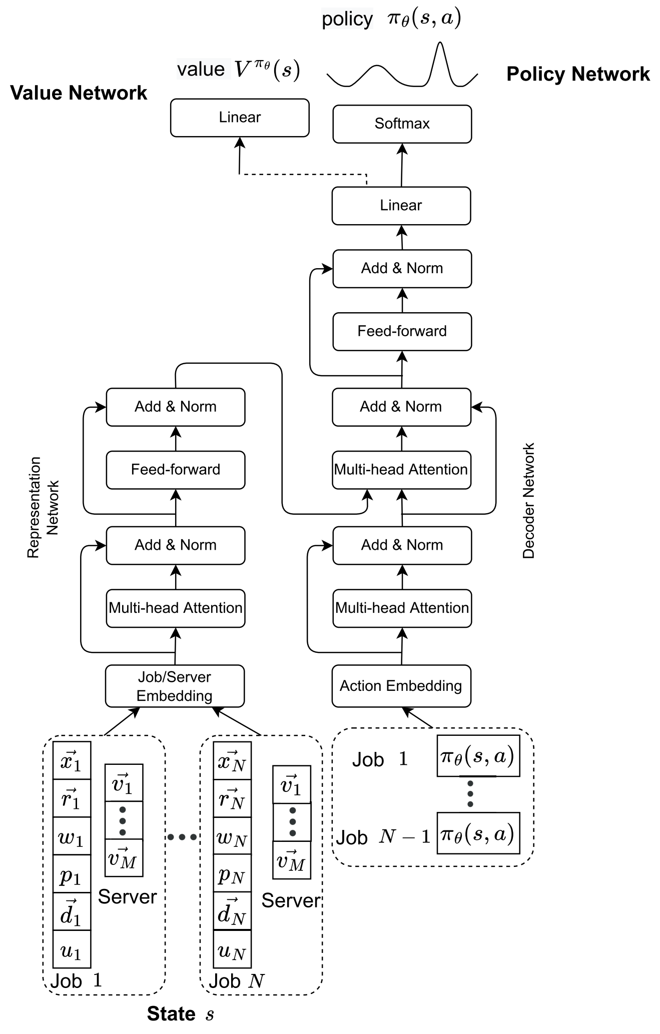


Fig. 7. DRL architecture.

(1) $\vec{x}_n$, an L-dimensional binary vector encoding the ML models trained by the job n, where 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). For simplicity, each vector $\vec{x}_n$, is a one-hot encoding of job n's model [39]. The same ML model, *e.g.*, a DNN of the same architecture and mini-batch size but possibly different learning rates and total numbers of training epochs, uses the same encoding. For example, if there are 3 models in total and 3 concurrent jobs using each of the models respectively, then $\vec{x}_0 = [1,0,0]$, $\vec{x}_1 = [0,1,0]$, $\vec{x}_2 = [0,0,1]$. There exist other possible encoding methods, *e.g.*, feature embedding [53], and we leave the exploration of more efficient encoding approaches as a future work.

(2) $\vec{r}_n$, a $2(1+K)$-dimension vector encoding worker/parameter server resource demands in the jobs, where K is the number of resource types to compose a worker or a parameter server. In each vector $\vec{r}_n$ for job n, 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 parameter servers requested by job n and each parameter server's resource composition.

For jobs with all-reduce architecture, we manually set the entry of parameter server in resource demands as zero, and then the DRL model can differentiate the communication architecture type based on the resource vector. For example, considering a job with 3 workers and 2 parameter servers, 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 parameter server needs no GPU but 1 CPU core. For an all-reduce job with only 2 workers, each requiring two types of resources (GPUs and CPU cores), $\vec{r}_n = [2, 2, 4, 0, 0, 0]$ represents that each worker in the job needs 2 GPUs and 4 CPU cores, and the resource demand entry for parameter server is defined as zero.

(3) w_n (p_n), one integer, in which the w_n (p_n) is the number of workers (parameter servers) allocated to the n^{th} job. For jobs with all-reduce architecture, the number of parameter servers is always zero. For example, $w_0 = 1$ and $w_1 = 2$ represents that 1 and 2 workers are allocated to 2 jobs, respectively. We manually set $p_n = 0$ if job n is using the all-reduce architecture.

(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. For example, a server with 8 available CPU cores and 2 available GPUs is encoded as $\vec{v}_m = [8, 2]$.

(5) $\vec{d}_n$, a $M \times 2$-sized vector encoding the placement of workers and parameter servers of each concurrent job n on the servers. For the n^{th} job, in its vector $\vec{d}_n$, the number of workers and the number of parameter servers of job n placed on server m ($m = 1, \ldots, M$), is on the $2m - 1^{\text{th}}$ and $2m^{\text{th}}$ position, respectively. For all-reduce jobs, the $2m^{\text{th}}$ position is always zero. Suppose sever 2 hosts 1 parameter server and 1 worker of job n and server 5 hosts 1 parameter server and 2 workers of job n among 6 servers; we have $\vec{d}_n = [0, 0, 1, 1, 0, 0, 0, 0, 2, 1, 0, 0]$.

(6) u_n, an integer, which indicates whether the n^{th} job is a newly arrived one or has been placed in previous scheduling intervals. For example, $u_0 = 0$ and $u_1 = 1$ represent that job 0 has already been placed before and job 1 is a newly arrived job in the current scheduling interval, respectively.

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 all feasible placement decisions of the all new jobs in a scheduling interval (recall we do not adjust placement of existing jobs), and then we can produce the placements of all jobs by one inference; however, this leads to an action space of exponential size, due to the exponentially many placement combinations of all workers and parameter servers in all jobs. A large action space may incur significant training time and worse results [39]. To expedite policy learning, we make placement decisions for newly arrived jobs one by one, and produce a sequence with placement decision for each new job. Specifically, we simplify the action definition, and our action space contains $2M$ actions as follows: (i) $(0, m)$, meaning placing one worker of the newly arrived job on server m, $\forall m \in [1, M]$; (ii) $(1, m)$, placing one parameter server of the newly arrived job on server m, $\forall m \in [1, M]$. To handle the case of using the all-reduce architecture, we manually set the probability of (ii) to zero, and rescale other non-zero probabilities such that their sum still equals 1.

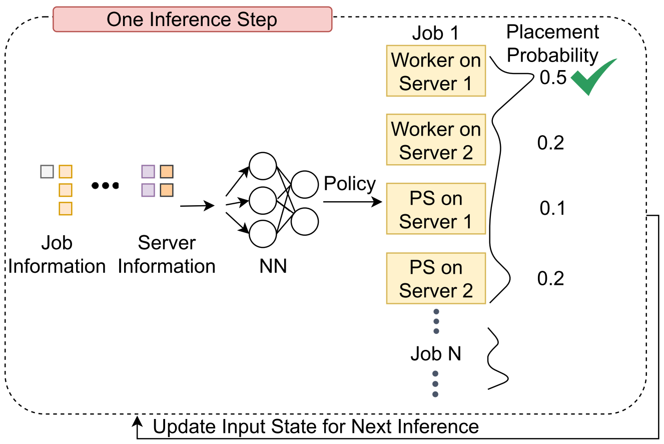


Fig. 8. Inference workflow.

In each scheduling interval, we allow multiple inferences over the NN, each selecting an action out of the action space for one of the newly arrived jobs. With each DNN inference, *Harmony* makes a decision for the placement of a parameter server or worker for a job, and then updates the state s accordingly (including placement and remaining available resources). Then in the next inference, *Harmony* will make placement decisions according to the updated input state and avoid co-location of jobs with intensive interference. After multiple inferences, we come up with a complete set of placement decisions for all workers and parameter servers in the new jobs (or the inferences stop when there are not enough resources to place any additional parameter server/worker). In this way, we use multiple inferences to effectively reduce action space by deciding $(0, m)$ and $(1, m)$ for all newly arrived jobs one by one; similar techniques have been adopted in [39], [49], and [19]. Fig. 8 shows an example of how to make a placement decision of one job through one inference, given job and server information.

Reward. We target average job completion time minimization by training the policy NN, which learns how to improve resource utilization and mitigate inter-job interference. 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. The significant feedback delay of the reward is unacceptable for training, since the delayed reward provides little guidance to improve the early decisions. Further, the 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 one scheduling interval (*e.g.*, 20 minutes):

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

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.

Representation network. The representation network takes the state of all concurrent jobs and server status as input at each scheduling interval, extracts features, and generates a representation (*i.e.*, a vector with a smaller size), which is then used in the decoder network to produce scheduling decisions. The main challenge is that the number of concurrent jobs is unknown beforehand, and may change over time. However, most NN structure, such as feed-forward NN, requires to have an input with a fixed size. A straightforward way to use feed-forward NN to deal with input of non-fixed size is to set an upper bound of the concurrent number of jobs and use padding in the input sequence, *e.g.*, put the entries as 0 if the job in that entry does not exist. This works if the actual number of concurrent jobs is smaller than the pre-defined upper bound. However, zero-padding causes lots of redundant information in the input state when the number of concurrent jobs is much smaller than the pre-defined job number upper bound. Similarly, if the number of the actual number of concurrent jobs is larger than the pre-defined upper-bound, we need to exclude some jobs, which leads to a loss in input information. To enable encoding of variable length input, we adopt the encoder part in Transformer [22] to encode the job and server information into a sequence of fix-sized matrices, where the attention model helps capture the correlation among different jobs in the input sequence. Specifically, the job set is orderless, and sequences with different job orders should have the same output policy distribution. Different from recurrent neural networks such as long short-term memory [48], the attention model is agnostic with the position of jobs in the input sequence and hence will not be impacted by the job order. We also adopt the residual network as the part of Transformer to avoid the problem of vanishing gradients. The output sequence of Transformer encoder is returned as the representation.

Decoder network. The decoder network is used to analyze the encoded sequence from the representation network and produce the placement decision one by one for each newly arrived job. The generated distribution for placement of other concurrent jobs are fed to the decoder as the decoder input, with the attention operation to deal with the impact of the placement decisions of other concurrent jobs. By utilizing the attention mechanism, it is possible for the decoder to capture global information rather than solely to infer based on one job placement decision. The decoder input processed by attention, together with the output of the representation network, is then aggregated to a decoder with a few hidden fully connected layers with the ReLU [54] function for activation, which is then connected to the final output layer. The final output layer uses the softmax function [55] as the activation function, and produce a sequence of decisions for each unscheduled job one by one. To respect server resource capacities, in the output layer of the NN, we mask the invalid actions, which deploy a worker or parameter server 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.

NN model. As illustrated in Fig. 7, the state of each job and server is connected to a fully connected layer (*i.e.*, Job/Server Embedding block in Fig. 7) first as embedding, and then they are connected to a representation network for encoding. With embedding, the NN can extract features from each job or each server as pre-processing. The pre-processing can also better differentiate the input sequence when the entries in the input sequence are similar. The pre-processed states of concurrent jobs are fed into the representation network one-by-one, and the representation network is learned similarly to a sequence learning process [56]. The representation network will be trained together with the decoder network in an end-to-end manner. We also feed the placement decisions of previous jobs (*e.g.*, Job 1 to Job $N-1$ in Fig. 7) into the decoder network, since the performance of the current placement decision (*e.g.*, for Job N in Fig. 7) is related to previous placement decisions we made within the current scheduling interval.

B. DRL Model Training

We apply the REINFORCE algorithm [57] 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 [57]:

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

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 value [57] 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 [58]. To reduce the variance, we improve the basic policy gradient training with the actor-critic algorithm [47]. 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 [58]. 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 [58].

2) Job-Aware 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 [39], [58]. One approach to encourage exploration is to add an entropy regularization term $\beta \cdot \bigtriangledown_{\boldsymbol{\theta}} H(\pi_{\boldsymbol{\theta}}(s_i, \cdot))$ [58] 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 [47]. When performing each inference over the policy NN (for its training), with probability $1 - \epsilon$, we adopt the action (worker/parameter server 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 parameter server (specified in the action selected by the NN) on a machine with least capacity left (where the worker or parameter server can still be accommodated); load balancing places one worker or parameter server 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) Multi-Head Attention: The attention mechanism is designed to capture the dependency in the input job sequence, by computing the dot product of all pairs of job information with an attention function, and then computing a weighted sum of the information of concurrent jobs in the input sequence. One attention layer can map the dependency into one sequence, each with a smaller vector. Besides, we project the job information with different attention functions in parallel, yielding multi-dimensional output. Multi-head attention is designed to allow the representation network to jointly attend to information from different representation sub-spaces at different positions. We employ 8 parallel attention layers (*i.e.*, heads) in experiments.

4) 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 [41]. 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 [41] 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.

VI. 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: a sequence $s = (s_1, \ldots, s_N)$ where $s_n = (\vec{x}_n, w_n, p_n, \vec{d}_n)$, including the following: (i) $\vec{x}_n$, job n's model type. (ii) w_n and p_n, allocated numbers of workers and parameter servers of the n^{th} job. (iii) $\vec{d}_n$, job n's placement on each server. The resource demands of a worker and a parameter server 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), based on the current placement indicated in $\vec{d}_n, \forall n \in [N]$). We do not directly output predicted completion time of the jobs, due to the following: throughout a job's execution, the placement on servers where its workers/parameter servers are

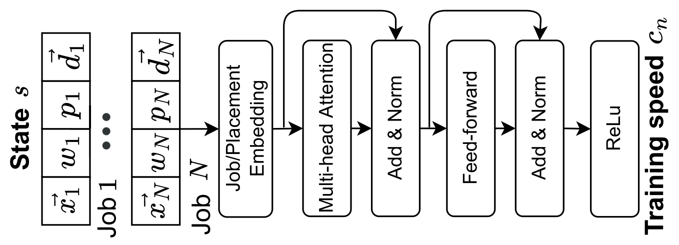


Fig. 9. Reward prediction model architecture.

deployed may change, due to arrival of new jobs, which affects job completion time; therefore, given the current job placement in the cluster, it is more reasonable to predict the training speed under the current placement, rather than the eventual job completion time. The input to the NN also has a non-fixed input size, and we adopt a sequence-to-sequence Transformer model to process input jobs and produce a sequence, *i.e.*, the predicted training speeds of the concurrent jobs. The input state of each job is connected to a sequence of hidden fully connected layers for embedding (Job/Placement Embedding block in Fig. 9), and then a Transformer-based encoder before a linear output layer. Similar to Sec. V-A, we also use the attention mechanism to capture the correlation among concurrent jobs. We do not include the decoder part of Transformer as there is no dependency in the output sequence, and we can produce the whole output sequence at once. 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 [59]). The detailed architecture can be found in Fig. 9.

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 a loss function, which is 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%).

VII. Performance Evaluation

We evaluate *Harmony* using testbed experiments, and simulation to validate the performance in a larger scale.

A. Implementation

Scheduler on Kubernetes. We implement *Harmony* using python on Kubernetes 1.7 [23] as a custom scheduler. We run workers and parameter servers on docker containers. Training datasets of jobs are stored in HDFS 2.8 [60]. 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 for these jobs by doing inference on the trained policy network. Each inference takes 4ms in a GTX 1080Ti GPU. The Kubernetes agents start the workers/parameter servers 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 first embedding layer has the same number of nodes as the size of each input entry in the sequence or decoder input. The representation network has 2 hidden layers with 128 nodes, and 8 attention heads. The decoder NN has 2 hidden layers with 128 neurons in the each 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 [61] 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 [1]. The reward NN has 2 hidden-encoder layers with 128 neurons in each hidden layer, and 8 attention heads in Transformer structure. We train it using Adam optimizer [61] with a variable learning rate following [22] and a batch size of 32 samples. We also apply batch normalization for reward NN to accelerate convergence [59]. 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/parameter server configurations are sufficient to saturate resource capacity of our cluster. We treat all-reduce as a special case that a worker and a parameter server are put together as a bundle. We run each job for about 5 minutes and calculate the average training speed. Due to the iterativeness of model training, running for 5 minutes should be sufficient to give us a rough idea of the training speed.

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. We deploy Kubernetes 1.7 as the cluster manager.

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/or parameter servers randomly in $[1, 3]$ to generate a job variant. With 0.5 probability, we choose parameter server architecture, and with 0.5 probability, we choose all-reduce architecture. For jobs training large datasets (*e.g.*, ImageNet [62]), we downscale the datasets so that the training can be finished in a reasonable amount of time. The job length follows a skewed distribution extracted from a production cluster [19], where there are a large number of short jobs and

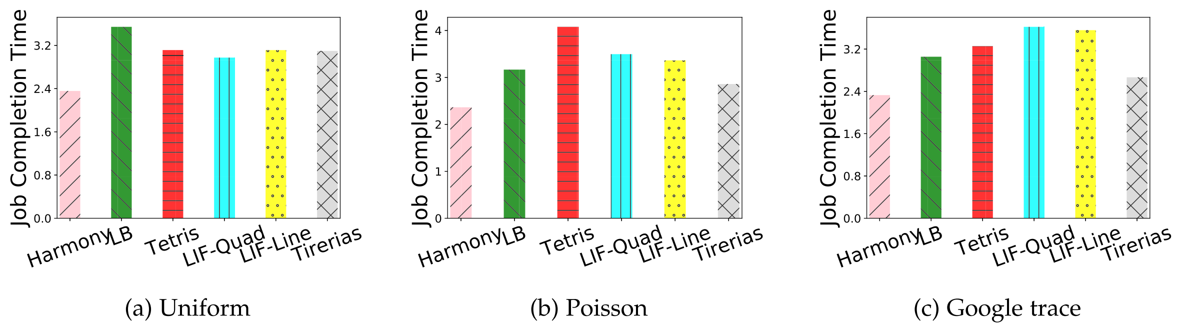


Fig. 10. Performance comparison under different job arrival patterns.

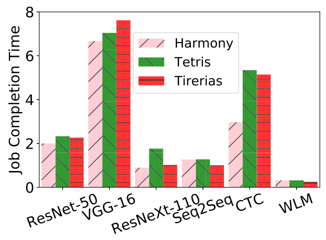


Fig. 11. Performance comparison of each job type.

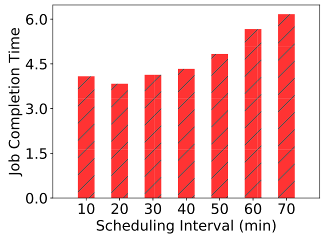


Fig. 12. Sensitive analysis of scheduling interval.

a few long jobs. We use the same sequence of jobs (and their arrival time) for *Harmony* and the baselines.

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

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

– Tetris [46]: it uses multi-resource bin packing to place a worker/parameter server to avoid resource fragmentation.

– Tiresias [31]: it proposes 2D-Gittins index to estimate job completion time and uses 2D-LAS (a SRTF variant) to minimize average job completion time for deep learning jobs. Tiresias also considers job placement by using spreading or consolidation policy according to job characteristics.

– Least Interference First (LIF-Line, LIF-Quad) [13], [63]: 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/parameter server, the algorithm calculates an interference score (*i.e.*, performance slowdown of all workers/parameter servers on a server) for each server and selects the server with the least interference.

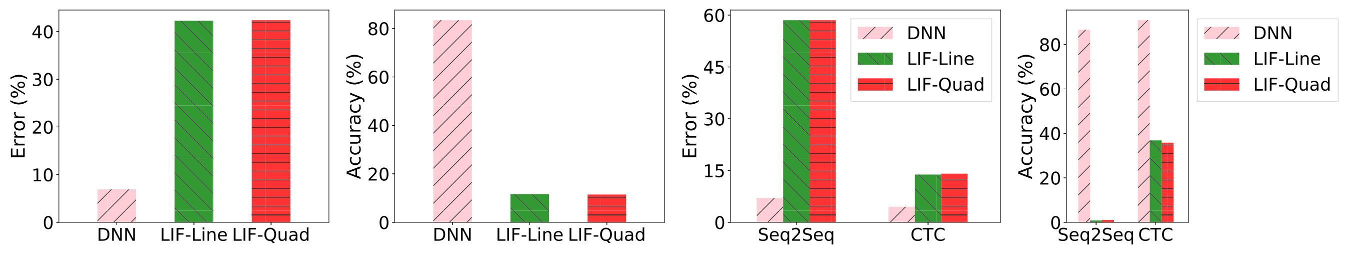
Metrics. We use the average job completion time as the main performance metric.

C. Performance

Fig. 10 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 [64], with downscaled arrival rates. *Harmony* improves average job completion time (in terms of number of intervals) by 16%-42% 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 interference among jobs. Tiresias does not consider CPU or IO interference in the placement policy. For network interference, it uses a threshold to determine whether a job should be consolidated or not according to the skewness in tensor distributions across parameter servers [31]. How to determine a good threshold is also difficult.

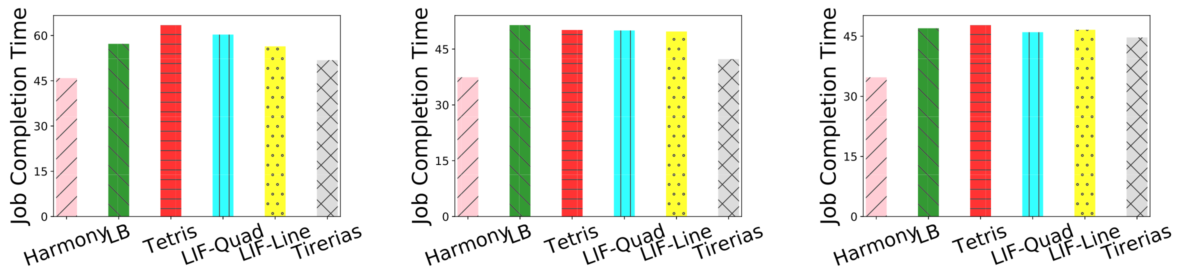
Fig. 11 shows the breakdown of the job completion time for different types of jobs. We find that *Harmony* can optimize all jobs, though there is not much difference for some jobs like Seq2Seq. We also conduct a sensitive analysis by varying job scheduling interval from 10 minutes to 70 minutes in Fig. 12. With a smaller job scheduling interval, we have fewer jobs in each scheduling interval and could miss some optimization opportunities. With a larger job scheduling interval, we can optimize an overall completion time with more jobs in a batch, but job waiting time is longer. The sweet spot of the scheduling interval in our system in 20 minutes.

We also compare the reward model with interference model in LIF-Line and LIF-Quad. We shuffle the collected trace (29264 samples) and split it into training dataset (90%) and test dataset (10%). Fig. 13(a) shows that our model achieves 6.9% relative error, much lower than that of the interference models in LIF-Line and LIF-Quad. We also compute the accuracy for each solution in Fig. 13(b), by finding the percentage of



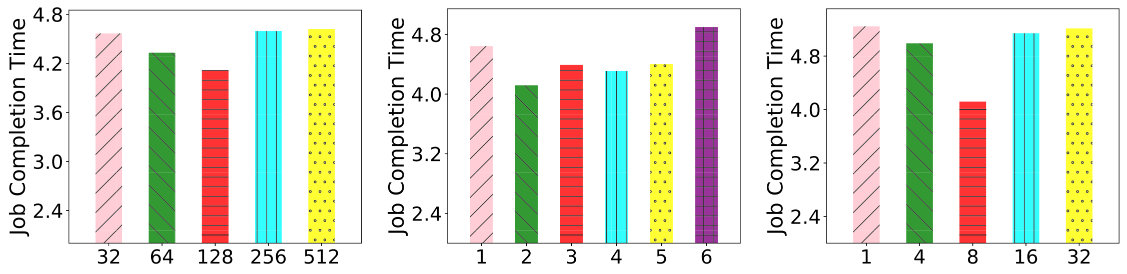
(a) Overall error (b) Overall accuracy (c) Error for jobs training Seq2Seq and CTC (d) Accuracy for jobs training Seq2Seq and CTC

Fig. 13. Training speed prediction.



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

Fig. 14. Performance comparison under different job arrival patterns over 30 machines.



(a) # of neurons (b) # of hidden layers (c) # of attention heads

Fig. 15. Deep dive.

samples whose relative errors are within 10%. We can find that our model can ensure that for over 83% of predictions, their relative error is less than 10%. Further, Fig. 13(c) shows the relative errors incurred and accuracy 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 (due to low pre-processing overhead), and carries out computation mostly on GPU; LIF builds interference models for CPU and network sharing only. The accuracy shown in Fig. 13(d) is similar.

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 [13], [63], as shown in Fig. 14. We use the simulated traces to train reward prediction NN and our DRL NN, and set the required number of workers and/or parameter servers randomly in $[1, 10]$. We observe that *Harmony* can still reduce average job completion time by at least 21%, as compared to the baselines.

D. Deep Dive

We next evaluate our detailed design of *Harmony*.

Number of neurons. We fix the number of hidden layers to 2 in both encoder and decoder and vary the number of neurons in the hidden layers. We train all neural networks until convergence over the same training set. Fig. 15(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 128 in the hidden layers in both Transformer encoder and decoder, and vary the number of hidden layers. As shown in Fig. 15(b), the NN with 2 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.

Number of attention heads. We fix the number of neurons to 128 with two hidden layers in both Transformer encoder and decoder, and vary the number of attention heads from 1 to 32. As shown in Fig. 15(c), the NN with 8 attention heads performs the best. With fewer heads, there are not enough parameters to encode the input; with more heads, it generally

TABLE II
EFFECTIVENESS OF TRAINING TECHNIQUES

Without	Avg. Job Completion Time (intervals)	Slowdown (%)
-	4.119	0
Value network	5.48	33.04%
Exploration	5.583	35.54%
Multi-head attention	5.246	27.36%
Experience replay	5.137	24.71%

increases the complexity of NN, requiring more trials to train all NN parameters to suitable values, which leads to unstable training progress.

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 33.04% 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.*, 35.54% 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.

Multi-head attention. We change the number of heads to one and examine how the number of attention layers influences the performance. From Table II, we see that the performance is worse than with 8 attention heads (*i.e.*, 27.36% slowdown). The reason is that without enough attention layers, it is hard for the representation network to keep necessary information of the input state.

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 24.71%, indicating that disrupting the order of samples and using samples among different scheduling intervals to update NN is critical for our 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. We consider online job arrival with either parameter server or all-reduce architecture. 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 adopt a transformer-based structure to mitigate unknown and dynamic job arrival sequence in both DRL model and reward model. We believe that our reward prediction module design is general, and applicable to other DRL problems where historical traces are not sufficient. We conduct a comprehensive empirical study to evaluate the performance of *Harmony* in different scenarios. Evaluation on a Kubernetes cluster shows that *Harmony* outperforms representative scheduling policies by 16%-42%, in reducing average job completion time in the cluster.

REFERENCES

[1] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. USENIX OSDI*, 2016, pp. 265–283.
[2] T. Chen *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," in *Proc. NeurIPS Workshop Mach. Learn. Syst. (LearningSys)*, 2016, pp. 1–6.
[3] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. NeurIPS*, 2019, pp. 8026–8037.
[4] M. Li, "Scaling distributed machine learning with the parameter server," in *Proc. USENIX OSDI*, 2014, pp. 583–598.
[5] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *Proc. USENIX OSDI*, 2014, pp. 571–582.
[6] A. Sergeev and M. Del Balso, "Horovod: Fast and easy distributed deep learning in TensorFlow," 2018, *arXiv:1802.05799*.
[7] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. ACM EuroSys*, 2015, pp. 1–17.
[8] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. USENIX NSDI*, 2011, pp. 295–308.
[9] Y. Kim, "Convolutional neural networks for sentence classification," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1746–1751.
[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. NeurIPS*, 2012, pp. 1097–1105.
[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. ICLR*, 2015, pp. 1–14.
[12] V. K. Vavilapalli *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, Oct. 2013, pp. 1–16.
[13] X. Bu, J. Rao, and C.-Z. Xu, "Interference and locality-aware task scheduling for MapReduce applications in virtual clusters," in *Proc. ACM HPDC*, 2013, pp. 227–238.
[14] F. Xu, F. Liu, P. Yin, and H. Jin, "Network-aware task assignment for MapReduce applications in shared clusters," *J. Internet Technol.*, vol. 16, no. 2, pp. 325–333, 2015.
[15] A. Gupta, L. V. Kale, D. Milojicic, P. Faraboschi, and S. M. Balle, "HPC-aware VM placement in infrastructure clouds," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Mar. 2013, pp. 11–20.
[16] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.
[17] D. Silver *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017.
[18] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 270–288.
[19] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Meng, and W. Lin, "DL2: A deep learning-driven scheduler for deep learning clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 8, pp. 1947–1960, Aug. 2021.
[20] Y. Gong, B. Li, B. Liang, and Z. Zhan, "Chic: Experience-driven scheduling in machine learning clusters," in *Proc. Int. Symp. Qual. Service*, Jun. 2019, pp. 1–10.
[21] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 197–210.
[22] A. Vaswani *et al.*, "Attention is all you need," in *Proc. NeurIPS*, 2017, pp. 6000–6010.
[23] (2018). *Kubernetes*. [Online]. Available: https://kubernetes.io
[24] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *J. Parallel Distrib. Comput.*, vol. 69, no. 2, pp. 117–124, Feb. 2009.
[25] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proc. 13th EuroSys Conf.*, Apr. 2018, pp. 1–14.

[26] (2018). *PaddlePaddle*. [Online]. Available: http://www.paddlepaddle.org/
[27] P. Sun, Y. Wen, N. B. D. Ta, and S. Yan, "Towards distributed machine learning in shared clusters: A dynamically-partitioned approach," in *Proc. IEEE Int. Conf. Smart Comput. (SMARTCOMP)*, May 2017, pp. 1–6.
[28] Y. Bao, Y. Peng, C. Wu, and Z. Li, "Online job scheduling in distributed machine learning clusters," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2018, pp. 495–503.
[29] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "SLAQ: Quality-driven scheduling for distributed machine learning," in *Proc. Symp. Cloud Comput.*, Sep. 2017, pp. 390–404.
[30] W. Xiao *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. USENIX OSDI*, 2018, pp. 595–610.
[31] J. Gu *et al.*, "Tiresias: A GPU cluster manager for distributed deep learning," in *Proc. USENIX NSDI*, 2019, pp. 485–500.
[32] A. Or, H. Zhang, and M. J. Freedman, "Resource elasticity in distributed deep learning," in *Proc. MLSys*, 2020, pp. 400–411.
[33] K. Mahajan *et al.*, "THEMIS: Fair and efficient GPU cluster scheduling," in *Proc. USENIX NSDI*, 2020, pp. 289–304.
[34] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, "Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning," in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, pp. 1–16.
[35] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *Proc. USENIX OSDI*, 2020, pp. 481–498.
[36] M. Mihailescu, S. Sharify, and C. Amza, "Optimized application placement for network congestion and failure resiliency in clouds," in *Proc. IEEE 4th Int. Conf. Cloud Netw. (CloudNet)*, Oct. 2015, pp. 7–13.
[37] F. Xu, F. Liu, and H. Jin, "Heterogeneity and interference-aware virtual machine provisioning for predictable performance in the cloud," *IEEE Trans. Comput.*, vol. 65, no. 8, pp. 2470–2483, Aug. 2016.
[38] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.
[39] O. Vinyals *et al.*, "StarCraft II: A new challenge for reinforcement learning," 2017, *arXiv:1708.04782*.
[40] A. Mirhoseini *et al.*, "Device placement optimization with reinforcement learning," in *Proc. ICML*, 2017, pp. 2430–2439.
[41] Z. Xu *et al.*, "Experience-driven networking: A deep reinforcement learning based approach," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2018, pp. 1871–1879.
[42] N. Liu *et al.*, "A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 372–382.
[43] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," in *Proc. ICLR*, 2018, pp. 1–11.
[44] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *Proc. IEEE INFOCOM*, Apr. 2019, pp. 1288–1296.
[45] (2017). *MXNet Official Examples*. [Online]. Available: https://github.com/apache/incubator-mxnet/tree/master/example
[46] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 455–466.
[47] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998.
[48] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[49] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. 15th ACM Workshop Hot Topics Netw.*, Nov. 2016, pp. 50–56.
[50] (2018). *Microsoft Azure*. [Online]. Available: https://azure.microsoft.com/en-us/
[51] (2018). *Alibaba Cloud*. [Online]. Available: https://www.alibabacloud.com/zh
[52] Y. Chen, Y. Peng, Y. Bao, C. Wu, Y. Zhu, and C. Guo, "Elastic parameter server load distribution in deep learning clusters," in *Proc. 11th ACM Symp. Cloud Comput.*, Oct. 2020, pp. 507–521.
[53] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. ICLR*, 2013, pp. 1–12.
[54] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proc. ICML*, 2010, pp. 807–814.
[55] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.
[56] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. NeurIPS*, 2014, pp. 3104–3112.
[57] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. NeurIPS*, 1999, pp. 1057–1063.
[58] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," in *Proc. ICML*, 2016, pp. 1928–1937.
[59] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 770–778.
[60] (2014). *HDFS*. [Online]. Available: https://wiki.apache.org/hadoop/HDFS
[61] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. ICLR*, 2015, pp. 1–15.
[62] (2018). *ImageNet Dataset*. [Online]. Available: http://www.image-net.org
[63] R. C. Chiang and H. H. Huang, "TRACON: Interference-aware scheduling for data-intensive applications in virtualized environments," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2014, pp. 1–12.
[64] C. Reiss and A. Tumanov, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. ACM Symp. Cloud Comput.*, 2012, pp. 1–13.

Yixin Bao received the B.Eng. degree from the Department of Automation, Xi'an Jiaotong University, in 2015, and the Ph.D. degree from the Department of Computer Science, The University of Hong Kong, in 2020. Her research interests include cloud computing, machine learning systems, and online learning algorithms.

Yanghua Peng received the B.E. degree from the Department of Computer Science and Technology, Wuhan University, China, in 2015, and the Ph.D. degree from the Department of Computer Science, The University of Hong Kong, in 2020. His research interests include cloud computing, cluster scheduling, and machine learning systems.

Chuan Wu (Senior Member, IEEE) received the Ph.D. degree from the Department of Electrical and Computer Engineering, University of Toronto, Canada, in 2008. Since September 2008, she has been with the Department of Computer Science, The University of Hong Kong, where she is currently a Professor. Her current research interests include cloud computing, distributed machine learning systems, and intelligent elderly care technologies.