DL^2: A Deep Learning-Driven Scheduler for Deep Learning Clusters

Yanghua Peng, Yixin Bao, *Student Member, IEEE*, Yangrui Chen, Chuan Wu, *Senior Member, IEEE*, Chen Meng, and Wei Lin

Abstract—Efficient resource scheduling is essential for maximal utilization of expensive deep learning (DL) clusters. Existing cluster schedulers either are agnostic to machine learning (ML) workload characteristics, or use scheduling heuristics based on operators' understanding of particular ML framework and workload, which are less efficient or not general enough. In this article, we show that DL techniques can be adopted to design a generic and efficient scheduler. Specifically, we propose DL^2, a DL-driven scheduler for DL clusters, targeting global training job expedition by dynamically resizing resources allocated to jobs. DL^2 advocates a joint supervised learning and reinforcement learning approach: a neural network is warmed up via offline supervised learning based on job traces produced by the existing cluster scheduler; then the neural network is plugged into the live DL cluster, fine-tuned by reinforcement learning carried out throughout the training progress of the DL jobs, and used for deciding job resource allocation in an online fashion. We implement DL^2 on Kubernetes and enable dynamic resource scaling in DL jobs on MXNet. Extensive evaluation shows that DL^2 outperforms fairness scheduler (i.e., DRF) by 44.1 percent and expert heuristic scheduler (i.e., Optimus) by 17.5 percent in terms of average job completion time.

Index Terms—Deep learning, resource allocation, distributed training

1 Introduction

Recent years have witnessed the breakthrough of DL-based techniques in various domains, such as machine translation [1], image classification [2], and speech recognition [3]. Large companies have deployed ML clusters with tens to thousands of expensive GPU servers, and run distributed training jobs on one or different distributed ML frameworks (such as TensorFlow [4], MXNet [5], Petuum [6] and PaddlePaddle [7]), to obtain DL models needed for their AI-driven services. Even with parallel training, training a DL model is commonly very time and resource intensive. Efficient resource scheduling is crucial in operating a shared DL cluster with multiple training jobs, for best utilization of expensive resources and expedited training completion.

Two camps of schedulers exist in today's ML clusters. In the first camp, general-purpose cloud/cluster schedulers are applied, and possibly customized, for distributed ML job scheduling. For example, Google uses Borg [8] as its DL cluster scheduler; Microsoft, Tencent, and Baidu use custom versions of YARN-like schedulers [9] for managing DL jobs. Representative scheduling strategies used include First-In-First-Out (FIFO) and Dominant Resource Fairness (DRF) [10]. These schedulers allocate resources according to user specification and do not adjust resource allocation during training. As we will see in Section 2.1, setting the right amount of resources for a job is difficult and static resource allocation leads to resource under-utilization in the cluster.

In the second camp, recent studies have proposed white-box heuristics for resource allocation in ML clusters [11], [12], [13]. Typically they tackle the problem in two steps: set up analytical models for DL/ML workloads, and propose scheduling heuristics accordingly for online resource allocation and adjustment. Designing heuristics requires a deep understanding of ML frameworks and workloads, and the analytical model is tightly coupled with workload patterns (e.g., system threshold setting may mismatch job size distribution) and ML framework implementations (e.g., a new feature or optimization in evolving ML frameworks may invalidate the analytical model) [12]. Manually adapting the heuristic algorithms to workload or framework implementation changes is time-consuming. Further, the modeling does not consider interference or can not capture it accurately in a multi-tenant cluster, where in average 27.3 percent performance variation may happen (Section 2.1).

In this paper, we pursue a DL cluster scheduler that automates the learning process of scheduling policy and adapts to workload changes and ML framework implementations. Instead of relying on expert heuristics and explicit performance model, we investigate a black-box end-to-end approach enabled by modern learning techniques. We propose DL^2, a deep learning-driven scheduler for deep learning clusters, that elastically adjusts resource allocation to training jobs on the go. DL^2 learns resource allocation policies through experience using deep reinforcement learning (DRL): the policy neural network takes the current system

- *Yanghua Peng, Yixin Bao, Yangrui Chen, and Chuan Wu are with the University of Hong Kong, Hong Kong, China. E-mail: {yhpeng, yxbao, yrchen, cwu}@cs.hku.hk.*
- *Chen Meng is with NAOC, Beijing 100012, China. E-mail: mengchen.cas@foxmail.com.*
- *Wei Lin is with Alibaba Inc., Hanzhou, Zhejiang 311121, China. E-mail: weilin.lw@alibaba-inc.com.*

Manuscript received 9 Feb. 2020; revised 14 Jan. 2021; accepted 15 Jan. 2021. Date of publication 19 Jan. 2021; date of current version 19 Feb. 2021. (Corresponding author: Yanghua Peng.) Recommended for acceptance by Y. Yang. Digital Object Identifier no. 10.1109/TPDS.2021.3052895

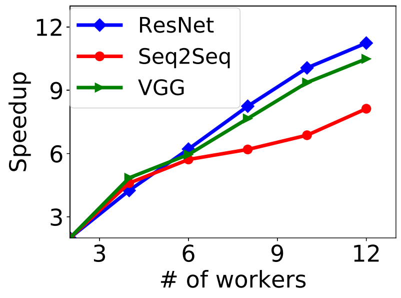


Fig. 1. Training speed-up with different worker/PS numbers.

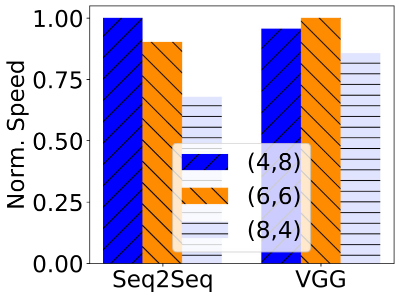


Fig. 2. Training speed under different PS-to-worker ratios

state as input, produces resource allocation decisions for all the current training jobs and gradually improves the decisions based on feedback. However, merely applying off-the-shelf RL algorithms to schedule does not produce high-quality decisions, and careful design according to the problem nature is in need.

Existing DRL applications in resource scheduling scenarios [14], [15], [16] (Section 8) use simulators to generate training data for offline training, and apply trained models for resource scheduling in a live system. The core of such a simulator is typically an explicit performance model as mentioned above, and hence the inaccuracy of the simulator may lead to a low-quality trained model. Instead of extensive offline training over large simulation, DL^2 takes a different approach: we bootstrap the model using minimal offline supervised learning with any available historical job traces and decisions of any existing scheduling strategy employed in the cluster; then we use online training with feedback from ongoing decision making in a live system, with carefully designed techniques to guide model convergence to high-quality decisions, which minimize average job completion time in the cluster.

In summary, we make the following contributions in DL^2:

▷ In contrast to previous DL scheduling approaches that require expert knowledge of workloads and ML framework implementations, DL^2 adopts a more generic design, i.e., using DRL to automatically adapt to environment changes and improve scheduling policy. To avoid inaccurate feedback from performance model based simulation, we adopt online training with real feedback from resource allocation (Section 2).

▷ To avoid poor decisions at the beginning of online RL, we apply past decisions made by an existing scheduler in the DL cluster in a preparatory offline supervised learning stage. Our approach enables a smooth transition from an existing scheduler, and automatically learns a better scheduler beyond the performance level of the existing one (Section 3). To optimize online RL particularly for DL job scheduling, we propose job-aware exploration for efficient exploration in the action space (Section 4).

▷ We design and implement elastic scaling in MXNet [5], to achieve dynamic worker/parameter server adjustment (Section 5). We integrate DL^2 with Kubernetes [17], and evaluate DL^2 using testbed experiments and controlled simulations, driven by DL job traces collected from a production DL cluster. Evaluation results show that DL^2 significantly outperforms representative schedulers in various scenarios, e.g., 44.1 percent improvement in average job completion time as compared to the widely adopted DRF scheduler. We also demonstrate DL^2's overhead and generality (Section 6). We have open-sourced DL^2 [18] in GitHub.

2 Background and Motivation

2.1 Motivation

We focus on the parameter server (PS) architecture [19], which is widely adopted in distributed ML learning frameworks for parallel training, such as in MXNet [5], TensorFlow [4], PaddlePaddle [7] and Angel [20]. The typical workflow for a user to train a model in a DL cluster is as follows: The user specifies how many PSs and workers she/he wishes to use and the amount of resources (e.g., GPU, CPU) each PS/worker needs, and then submits the job to the scheduler (e.g., Borg [8], YARN [9], Mesos [21]). The scheduler allocates PSs and workers to the job according to both user demand and its scheduling strategy, and the allocated resources then remain fixed over the entire training course of the job. This workflow has two limitations.

Difficulty in Setting the Right Worker/Ps Numbers. How does a job's training speed improve when more PSs and workers are added to the job? We train 3 classical models, i.e., ResNet-50 [22], VGG-16 [23] and Seq2Seq [24] and measure their training speeds (in terms of the number of samples trained per unit time), when increasing the number of workers and keeping the number of PSs equal to the worker number. In Fig. 1, the speed-up is calculated by dividing the training speed achieved using multiple workers by the training speed obtained using one worker. We observe a trend of decreasing return, i.e., adding PSs/workers does not improve the training speed linearly, due to increasing communication overhead.

On the other hand, is an equal number of PSs and workers (as a general rule of thumb) always the best? We fix the total number of PSs and workers to be 12 and measure the training speed of two models under different combinations of PS/worker numbers (i.e., 4:8, 6:6, 8:4) [12]. Fig. 2 shows that Seq2Seq achieves the highest training speed when there are 4 PSs and 8 workers, while VGG-16 is trained fastest with 6 PSs and 6 workers.

From the above, we see that it is challenging to reason about which job will have the largest marginal gain from extra resources and what the best PS-to-worker ratio is, as they are affected by many factors, e.g., allocated resources, models. Existing schedulers side-step this problem and leave it to the user to decide how many PSs/workers to use.

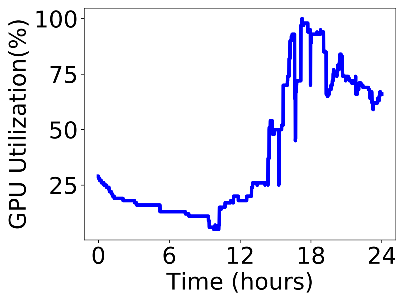


Fig. 3. GPU utilization in a production DL cluster.

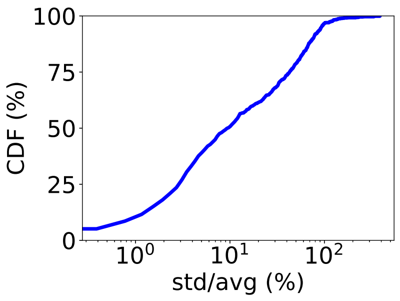


Fig. 4. Variation of training completion time.

Static Resource Allocation. The GPU cluster resources are often not fully utilized: when a training job is completed, the resources it releases (e.g., expensive GPUs) may become idle, rather than being exploited by remaining jobs that are still running. Fig. 3 shows the GPU utilization (i.e., the number of allocated GPUs divided by the total number of GPUs) during a 24-hour interval in a production DL cluster with about 1000 P100 GPU cards (company name removed due to anonymity requirement), whose job traces will be used in our evaluation (Section 6). We see that the GPU utilization level varies significantly over time, providing an opportunity for dynamic resource scaling out/in in training jobs when cluster load is low/high.

We advocate dynamic adjustment of worker/PS numbers in jobs over time, to maximally utilize resources in the DL cluster to expedite job completion. We further do not require users to submit the number of workers/PSs they want to use in their jobs (who nonetheless may not be at the best position to decide that), but will decide the best worker/PS numbers for users at each time based on both global resource availability and individual jobs' performance.

Expert Heuristics. There have been existing studies which explicitly model detailed relationship between the training speed and resources within jobs, and design scheduling heuristics based on the resource-speed model, e.g., SLAQ [11], Optimus [12] and OASiS [13]. They have two limitations.

First, in order to derive an accurate performance model, the modeling process is coupled tightly with ML framework implementations or workload patterns. Re-modeling is time-consuming (at least weeks [25]) and is often needed when the ML framework improves (e.g., adding new features or adopting optimization) or parameter setting mismatches workload pattern (e.g., job size distribution). For example, Optimus models computation and communication as two separate procedures during one training step; its model needs to be rebuilt when new features are incorporated into ML frameworks, e.g., overlapping backward computation with communication, gradient compression [5].

Second, explicit performance models are built without considering interference in multi-tenant GPU clusters. For example, SLAQ [11] and Optimus [12] assume no network congestion on PSs, and OASiS [13] and Optimus [12] assume that the available bandwidth is a constant. However, we observe that the speed for training the same model may change significantly. Fig. 4 shows the performance variation (i.e., the standard deviation of completion time of a training job divided by average completion time of the job over its multiple runs) of 898 DL jobs from the production ML cluster trace. The average variation is 27.3 percent and the variation for some jobs (3.5 percent of all jobs) even exceeds 100 percent. Besides, explicitly modeling interference among ML jobs is also difficult [26], as adding an additional dimension (network structure, runtime isolation, etc.) increases complexity.

In contrast to white-box heuristics, we resort to a generic black-box approach and design an RL-based resource scheduler: it automatically adapts to environment changes and learns end-to-end resource allocation policy without requiring expert heuristics and without explicitly modeling the ML framework, the workload, and the interference.

2.2 Deep Reinforcement Learning

DRL has been widely used for sequential decision making in an unknown environment, where the agent learns a policy to optimize a cumulative reward by trial-and-error interactions with the environment [27]. In each iteration, the agent observes the current state of environment and chooses an action based on the current policy. The environment moves to a new state and reveals the reward, and the policy is updated based on the received reward.

Existing DRL-based schedulers for resource allocation [14], [15], [16], [28] generate a large amount of traces for offline DRL model training, typically by building an explicit resource-performance model for jobs and using it to estimate job progress based on the allocated resources, in the offline simulation environment. The need for model rebuilding (due to ML system changes) and inaccuracy (due to interference) of the performance model degrade the quality of the DRL policy learned (see Fig. 9). Another possibility is to use available historical traces for offline DRL training. However, due to the large decision space of resource allocation (exponential with the amount of resources), historical traces usually do not include feedback for all possible decisions produced by the DRL policy [26].

Therefore, instead of offline training in a simulated environment, we advocate online RL in the live cluster and exploit true feedback for resource allocation decisions produced by the DRL agent, to learn a good policy over time. Pure online learning of the policy network model from scratch can result in poor policies at the beginning of learning (see Fig. 10). To avoid poor initial decisions and for the smooth transition from an existing scheduler, we adopt offline supervised learning to bootstrap the DRL policy with the existing scheduling strategy.

3 DL^2 Overview

DL^2 targets the best resource allocation policy in a live DL cluster, and minimize the average job completion time among all concurrent jobs. An overview of DL^2 is given in Fig. 5.

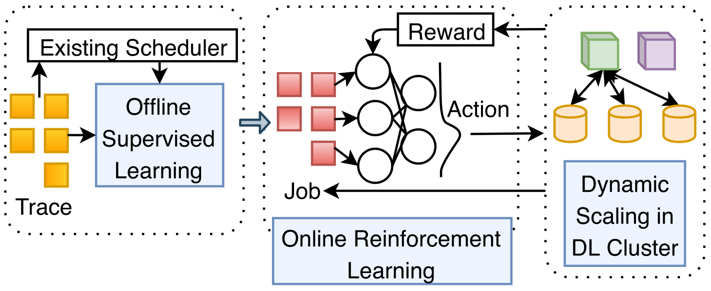

Existing Scheduler
Offline Supervised Learning
Trace
Reward
Action
Job
Online Reinforcement Learning
Dynamic Scaling in DL Cluster

Fig. 5. An overview of DL^2.

3.1 DL Cluster

In the DL cluster with multiple GPU servers, DL training jobs are submitted over time. Upon submission of a job, the user, i.e., job owner, provides her/his resource demand to run each worker and each PS, respectively, as well as the total number of training epochs to run. For example, a worker often requires at least 1 GPU, and a PS needs many CPU cores. The total training epoch number to achieve model convergence can be estimated based on expert knowledge or job history.

Depending on resource availability and training speeds, each job may run over a different number of workers and PSs from one time slot to the other (as decided by the scheduler). For synchronous training, to guarantee the same training result (model) while varying the number of workers, we adjust the mini-batch size of each worker, so that the total batch size in a job, as specified by the user, still remains unchanged [12], [29]. For asynchronous training, the mini-batch size of each worker remains the same while the number of workers varies (as the global batch size equals each worker's batch size).

3.2 DL^2 Scheduler

Offline Supervised Learning. For the warm-up, we use supervised learning to train the policy NN, to initialize scheduling policy. The historical job runtime traces collected from the cluster are used for training the NN to produce similar decisions as made by the existing scheduler. This step is a must due to the poor performance of applying online RL directly (see Fig. 10).

Online Reinforcement Learning. Online RL works in a time-slotted fashion; each time slot is a scheduling interval, e.g., 1 hour. At the beginning of a scheduling interval, the policy NN takes the information of all the concurrent jobs as input state, and produces the numbers of workers and PSs for each job. Jobs' training progress is observed at the end of each time slot, and used as the reward to improve the policy network.

Dynamic Scaling. Each job may need to adjust resources according to the decisions of policy NN in each time slot. To support dynamically adding or removing PSs/workers during training, we design and implement dynamic scaling in an ML framework, i.e., MXNet. It minimizes the resource scaling overhead while guaranteeing training correctness.

4 Learning Design

4.1 Policy Neural Network

We learn a neural network to produce resource allocation decisions to concurrent DL jobs through joint offline supervised and online reinforcement learning.

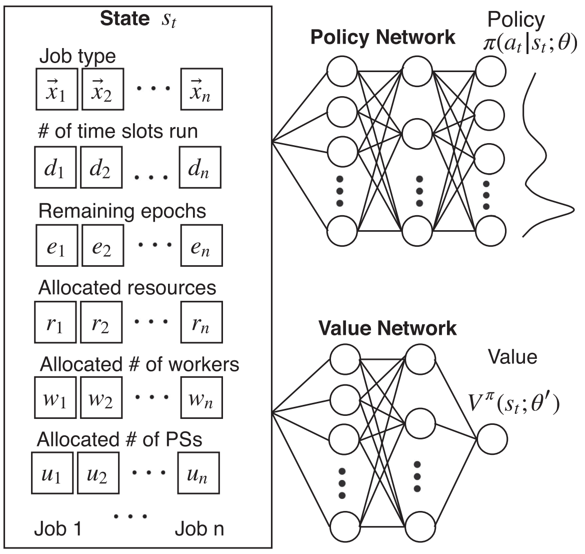


Fig. 6. Actor-critic reinforcement learning.

State. The input state to the policy NN is a matrix $s = (\boldsymbol{x}, \vec{d}, \vec{e}, \vec{r}, \vec{w}, \vec{u})$, including the following (Fig. 6):

- $\boldsymbol{x}$, a $J \times L$ matrix representing the DL models trained in the jobs, where J is an upper bound of the maximal number of concurrent jobs in a time slot that we are scheduling, and L is the maximal number of training job types in the cluster. We consider DL jobs training similar DNN architecture as the same type in our input. For example, fine-tuning jobs based on the same pre-trained model is common[1] and they can be treated as the same type. Each vector $\vec{x}_i$ in $\boldsymbol{x}$, $\forall i = 1, \ldots, J$, is an encoding of job i's type.
- $\vec{d}$, a J-dimensional vector encoding the number of time slots that each job has run in the cluster, for all jobs. For example, d_i is the number of time slots that job i has run.
- $\vec{e}$, a J-dimensional vector encoding the remaining number of epochs to train for each job. e_i is the difference between user-specified total training epoch number for job i and the number of epochs trained till current time slot.
- $\vec{r}$, a J-dimensional vector representing the proportion of dominant resource already allocated to each job in the current time slot. For example, r_i is the proportion of dominant resource (the type of resource that job i occupies most, divided by the overall capacity of that resource in the cluster) allocated to job i by resource allocation decisions already made through inferences in this time slot.
- $\vec{w}$ and $\vec{u}$, each of them is a J-dimensional vector where the ith item is the number of workers (PSs) allocated to job i in the current time slot.

  Information of concurrent jobs in different components of the state are ordered according to the jobs' arrival times. The input state does not directly include available resource capacities in the cluster;

1. Many computer vision jobs use pre-trained ResNet [22] model as initialization for training on a target dataset. Similarly, natural language understanding jobs use BERT [30] model to initialize training.

our scheduler can handle time-varying overall resource capacities in the cluster.

Action. The NN produces a policy $\pi : \pi(a \mid s; \vec{\theta}) \rightarrow [0, 1]$, which is a probability distribution over the action space. a represents an action, and $\vec{\theta}$ is the current set of parameters (i.e., the training weights of the hidden layers) in the NN. A straightforward design is to allow each action to specify the numbers of workers/PSs to allocate to all concurrent jobs; this leads to an exponentially large action space, containing all possible worker/PS number combinations. A large action space incurs significant training cost and slow convergence [31].

To expedite learning of the NN, we simplify the action definition, and allow the NN to output an action out of the following $3 \times J + 1$ actions through each inference: (i) $(i, 0)$, meaning allocating one worker to job i, (ii) $(i, 1)$, allocating one PS to job i, (iii) $(i, 2)$, allocating one worker and one PS to job i, (iv) a void action which indicates stopping allocating resources in the current time slot (as allocating more resources does not necessarily lead to higher training speed [12]). Since each inference only outputs an incremental amount of resources to be allocated to one of J jobs, we allow multiple inferences over the NN for producing the complete set of resource allocation decisions in each time slot: after producing one action, we update state s, and then use the NN to produce another action; the inference repeats until the resources are used up or a void action is produced. The void action indicates that further resource allocation no longer improves training speeds.

NN Architecture. The input state matrix s is connected to a fully connected layer with the ReLU [32] function for activation. The number of neurons in this layer is proportional to the size of the state matrix. Output from this layer is aggregated in a hidden fully connected layer, which is then connected to the final output layer. The final output layer uses the softmax function [33] as the activation function. The NN architecture is designed based on empirical training trials.

4.2 Offline Supervised Learning

In offline supervised learning, we use stochastic gradient descent (SGD) [34] to update parameters $\vec{\theta}$ of the policy NN to minimize a loss function, which is the cross entropy of the resource allocation decisions made by the NN and decisions of the existing scheduler in the traces [35]. The NN is repeatedly trained using the trace data, e.g., hundreds of times as in our experiments, such that the policy produced by the NN converges to the policy of the existing scheduler.

4.3 Online Reinforcement Learning

Reward. DL^2 targets average job completion time minimization in the entire cluster. 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 time slots later. The significant feedback delay of the reward is unacceptable for online RL, since the delayed reward provides little guidance to improve the early decisions. We design a per-time-slot reward to collect more reward samples through the job running processes, for more frequent RL model updates to expedite convergence. The per-timeslot reward is the sum of normalized number of epochs that the concurrent jobs have trained in this time slot, where the number of epochs trained in job i (t_i) is normalized over the overall number of epochs to train for the job (E_i):

$$r_t = \sum_{i \in [J]} \frac{t_i}{E_i}, \forall t = 1, \ldots, \tag{1}$$

The rationale is that the more epochs a job runs in a time slot, the fewer time slots it takes to complete, and hence maximizing cumulative reward amounts to minimizing average job completion time. The normalization is to prevent bias towards large jobs.

Policy Gradient-Based Learning. In online RL, the policy NN obtained through offline supervised learning is further trained using the REINFORCE algorithm [36], to maximize the expected cumulative discounted reward $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t]$, where $\gamma \in (0, 1)$ is the discount factor. The algorithm updates the policy network's parameters, $\vec{\theta}$, by performing stochastic gradient descent (SGD) on $\mathbb{E}[\sum_{t=0}^{\infty} -\gamma^t r_t]$. The gradient is:

$$\nabla_{\vec{\theta}} \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} -\gamma^t r_t\right] = \mathbb{E}_{\pi}\Big[- \nabla_{\vec{\theta}} \log(\pi(a \mid s; \vec{\theta}))Q(a, s; \vec{\theta})\Big], \tag{2}$$

where the Q value, $Q(a, s; \vec{\theta})$, is the RL reward feedback, i.e., the "quality" of the action a taken in a given state s following the policy $\pi(\cdot; \vec{\theta})$, calculated as the expected cumulative discounted reward to obtain after selecting action a at state s following $\pi(\cdot; \vec{\theta})$. To approximate the expectation, each Q value can be computed empirically using a mini-batch of training samples [34]. Each sample is a four-tuple, (s, a, s', r), 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 time slot t; the input state changes after each inference; we only observe the reward and update the NN once after all inferences in the time slot are done. We can obtain multiple samples in a time slot t, and set the reward in each sample to be the reward (1) observed after all inferences are done in t.

We further adopt a number of techniques to stabilize online RL, expedite policy convergence, and improve the quality of the obtained policy.

Actor-Critic. We improve the basic policy gradient-based RL with the actor-critic algorithm [37] (illustrated in Fig. 6), for faster convergence of the policy network. The basic idea is to replace Q value in Eq. (2) with an advantage, $Q(a, s; \vec{\theta}) - V^{\pi}(s, \vec{\theta})$, where $V^{\pi}(s, \vec{\theta})$ is a value function, representing the expected reward over the actions drawn using policy $\pi(a \mid s; \vec{\theta})$ at all times starting from time slot t. The advantage shows how much better a specific action is, as compared to the expected reward of taking actions according to $\pi(a \mid s; \vec{\theta})$ in the current state. Using the advantage in computing the policy gradients ensures a much lower variance in the gradients, such that policy learning is more stable.

The value function is evaluated by a value network, which has the same NN structure as the policy network except that its final output layer is a linear neuron without any activation function [37], and it produces the estimate of value function $V^{\pi}(s, \vec{\theta})$. The input state to the value network is the same as that to the policy network. We train the value network using temporal difference method [37].

Job-Aware Exploration. To obtain a good policy through RL, we need to ensure that the action space is adequately explored (i.e., actions leading to good rewards can be sufficiently produced); as otherwise, the RL may well converge to poor local optimal policy [31], [37]. We first adopt a commonly used entropy exploration method, by adding an entropy regularization term $\beta \nabla_{\vec{\theta}} H(\pi(\cdot \mid s; \vec{\theta}))$ in gradient calculation to update the policy network [37]. In this way, parameters of the policy network, $\vec{\theta}$, is updated towards the direction of higher entropy (implying exploring more of the action space).

During training, we observe a large number of unnecessary or poor explorations (e.g., allocating multiple workers but 0 PS to a job) due to unawareness of job semantics. To improve exploration efficiency, we adopt job-aware exploration combined with the ϵ-greedy method [27]. At each inference using the policy network, we check the input state: if the input state belongs to one of the poor states that we have identified, with probability $1-\epsilon$, we apply the resource allocation decisions produced by the policy network, and with probability ϵ, we discard the output, but adopt a specified action and observe the reward of this action.

The set of poor input states includes three cases: (i) there exists one job to be scheduled which has been allocated with multiple workers but no PS; (ii) there exists one job which has been allocated multiple PSs but no workers; (iii) there exists one job whose allocated numbers of workers (w) and PSs (u) differ too much, i.e., $w/u > threshold$ or $u/w > threshold$ (the threshold is 10 in our experiments). Our specified action upon each of these input states is: (i) allocate one more PS to that job; (ii) allocate one more worker to the job; (iii) allocate one more PS or worker to that job, to make its worker/PS numbers more even.

Experience Replay. It is known that correlation among the samples prevents convergence of an actor-critic model to a good policy [27]. In our online RL, the current policy network determines the following training samples, e.g., if the policy network finds that allocating more workers improves reward, then the following sample sequence will be dominated by those produced from this strategy; this may lead to a bad feedback loop, preventing the exploration of samples with higher rewards.

To alleviate correlation in the observed sample sequence, we adopt experience replay [38] in the actor-critic framework. Specifically, we maintain a replay buffer to store the samples collected from a large time span. At the end of each time slot, instead of using all samples collected during this time slot, we select a mini-batch of samples from the replay buffer to compute the gradient updates, where the samples could be from multiple previous time slots.

5 Elastic Scaling

Though node addition and deletion are supported in system design in the literature [19], [39], [40] , existing open-source distributed machine learning frameworks (e.g., TensorFlow [4], MXNet [5], Caffe [41]) do not support dynamic worker/PS adjustment in a running job. To adjust the number of workers/PSs in a job, a simple and general approach is checkpointing (e.g., Optimus [12]): terminate a training job and save global model parameters as a checkpoint image; then restart the job with a new deployment of PSs and workers, and the saved model parameters. Checkpointing and restarting add additional delay to the training process [40]. For example, it takes 1 minute to checkpoint the training of a DSSM model [42], and another 5 minutes to completely restore the training, due to data re-preprocessing before training starts. The overhead is significant when the frequency of resource scaling is high (e.g., every hour). The other approach is to resize resources without terminating the training process. As an example, we improve MXNet [5] to enable dynamic “hot” scaling.

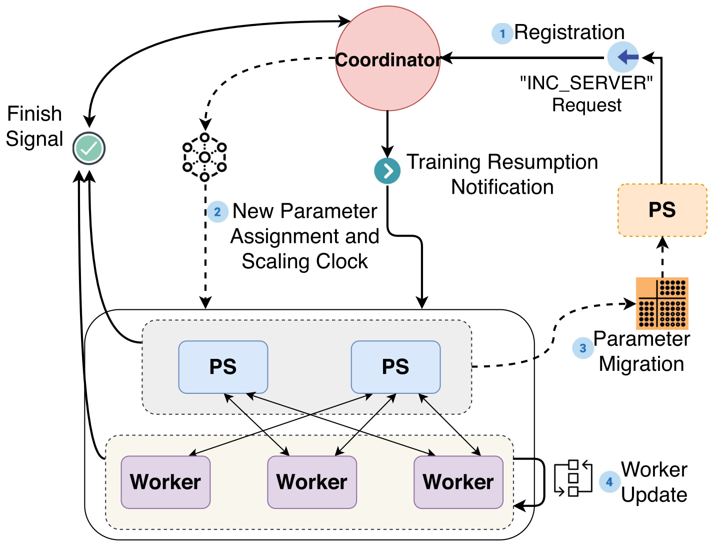


Fig. 7. Steps for adding one PS into a running job.

Challenges. In PS architecture, each PS maintains a subset of the parameters in the global model. In order to keep consistent model parameters, when the number of PSs changes, the global parameters need to be migrated among the PSs (for load balancing), and workers should be informed in time to send parameter updates to the correct PSs. When the number of workers changes, the new connections between new workers and the PSs should be established. The key challenges are: (1) *correctness*, i.e., a consistent copy of the global model parameters should be maintained while parameters are moved across the PSs, and workers always send gradients to correct PSs; (2) *high performance*, i.e., we should ensure that interruption to training is minimized and the PSs are load balanced.

Scaling Procedure. We add a *coordinator* module into MXNet. For each job, we launch a coordinator, which works with DL^2 scheduler to handle joining of new workers or PSs and termination of existing ones. We demonstrate our design using the case of adding a new PS into an existing job. The steps are shown in Fig. 7.

1) *Registration.* When a new PS is launched, it registers itself with the coordinator by sending an “INC_SERVER” request message. The PS will then receive its ID in the job, the global parameters it is responsible to maintain, and the current list of

workers and PSs to establish connections with. After that, the PS starts functioning, awaiting workers' parameter updates and further instructions from the coordinator (e.g., parameter migration).

2) *Parameter assignment.* Upon receiving a registration request, the coordinator updates its list of workers and PSs, and computes parameter assignment to the new PS. A best-fit algorithm is adopted: move part of the parameters on each existing PS to the new PS, such that all PSs maintain nearly the same number of parameters, while minimizing parameter movement across the PSs. In order to keep a correct and consistent copy of global model parameters (i.e., the number and value of parameters are not affected by scaling) when migrating parameters among PSs, we maintain a version counter for parameters. For PSs, the version counter is the number of parameter updates; for workers, the version counter is received from PSs when pulling updated parameters. To decide when PSs should migrate parameters, we calculate a scaling clock based on current version counter and round trip time between the coordinator and PSs/workers.

   The coordinator sends new parameter assignment among PSs and the scaling clock to all PSs and workers.

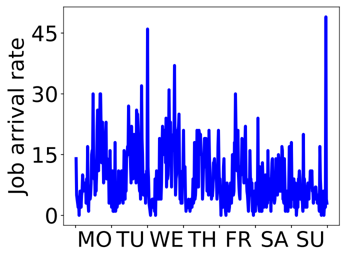
3) *Parameter migration.* At each PS, when the version counter of parameters reaches the scaling clock from the coordinator, the PS moves its parameters to the new PS according to the parameter assignment decisions received.[2] Once parameter migration among all PSs is completed, the coordinator notifies all workers to resume training.

4) *Worker update.* At each worker, once its version counter equals the scaling clock received from the coordinator, the worker suspends its push/pull operations and awaits notification for completion of parameter migration. Upon notification from the coordinator, the workers update their parameter-PS mapping, establish connections with the new PS, and resume training.

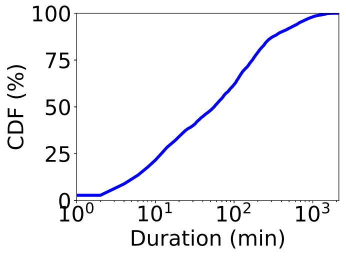
In case of removing a PS, the scheduler chooses the PS to be removed by keeping the load balanced among the physical machines. The chosen PS sends a removal request to the coordinator. Similar steps as 2)3)4) above are then carried out, except that parameters in the removed PS are moved to other PSs, using the best-fit algorithm.

To add a new worker into an existing job, the coordinator sends the current parameter-PS mapping in response to the worker's registration message. It also notifies all PSs the addition of the new worker for building connections. The worker starts operation after training dataset is copied. For worker removal, the scheduler chooses the worker to be removed by keeping the load balanced across physical machines. The coordinator receives a removal request from the worker, and then broadcasts it to all workers and PSs for updating their node lists. The coordinator should not be a scaling bottleneck as the removal or registration requests of a job are not frequent.

2. For asynchronous training, the PS buffers push or pull requests from workers and redirects them to the new PS.



(a) Job arrival rate in a week



(b) CDF of job duration

Fig. 8. Trace sketch.

6 Evaluation

6.1 DL^2 Implementation

We implement DL^2 as a custom scheduler on Kubernetes [17]. Cluster users' DL training scripts are written using standard MXNet APIs and run on the elastic MXNet framework (Section 5). Workers and PSs are running on Docker containers. Training data of jobs are stored in HDFS 2.8 [43]. The scheduler constantly queries cluster resources and job states (e.g., training speeds) and instructs deployment of a new job or resource adjustment in an existing job via Kubernetes API server. Mapping the cluster and job states to a scheduling decision (i.e., doing one inference) takes less than 3ms. The average end-to-end scheduling latency (i.e., the duration of job pending phase in Kubernetes) after integrating DL^2 into Kubernetes is about 700ms. The RL model is trained and updated once every scheduling slot. Since the duration of DL jobs is typically long (e.g., hours), DL^2 is sufficiently fast for large-scale DL clusters.

For each new job, DL^2 launches its coordinator, workers, and PSs on machines decided by the default placement strategy of the cluster (i.e., load balancing). The coordinator is informed of the workers and PSs in the job via Kubernetes API. When a worker/PS container is launched on a machine, an agent in the container starts execution. It queries the readiness of other containers of the same job via Kubernetes API and starts user-provided training scripts after all other containers are ready. The agent also monitors the training status, e.g., the number of trained steps, accuracy, and training speed.

6.2 Methodology

Testbed. Our testbed includes 13 GPU/CPU servers connected by a Dell Networking Z9100-ON 100GbE switch. Each server has one Intel E5-1660 v4 CPU, two GTX 1080Ti GPUs, 48GB RAM, one MCX413A-GCAT 50GbE NIC, one 480GB SSD, and one 4TB HDD. Each server runs Ubuntu 14.04 LTS and Docker 17.09-ce [44].

Trace. We use patterns from a 75-day real-world job trace collected from a large production DL cluster with a few thousands of GPUs and thousands of jobs, to drive our testbed experiments and simulation studies. Fig. 8a shows the job arrival rate (number of jobs arrived per time slot, i.e., 20 minutes) during a typical week. Fig. 8b shows the distribution of job duration: over a half of jobs run for more than an hour and some for days; the average job duration is 147 minutes.

Due to security and privacy concerns of the company, the job source code is not available, and we do not know job

TABLE 1
DL Jobs in Evaluation

Model	Application domain	Dataset
ResNet-50	image classification	ImageNet
VGG-16	image classification	ImageNet
ResNeXt-110	image classification	CIFAR10
Inception-BN	image classification	Caltech
Seq2Seq	machine translation	WMT17
CTC	sentence classification	mr
DSSM	word representation	text8
WLM	language modeling	PTB

details (e.g., model architecture). So we select 8 typical categories of ML models for experiments, from official MXNet tutorials [45], with representative application domains, different architectures and parameter sizes [45], as shown in Table 1. Each worker in different jobs uses at most 2 GPUs and 1-4 CPU cores, and each PS uses 1-4 CPU cores.

In both testbed experiments and simulations, the jobs are submitted to the cluster following the dynamic pattern in Fig. 8a (with arrival rates scaled down). Upon an arrival event, we randomly select a model from Table 1 and vary its number of training epochs (tens to hundreds) to generate a job variant, following job running time distribution of the real-world trace (scaled down). For models training on large datasets (e.g., ImageNet [46]), we downscale the datasets so that the training can be finished in a reasonable amount of time. In experiments, 30 jobs are submitted to run in our testbed; in simulations, 500 servers are simulated, and 200 jobs are submitted.

Training Setting. We use TensorFlow [4] to build and train the policy neural network of DL^2. The neural network is trained using Adam optimizer [47] with a fixed learning rate of 0.005 for offline supervised learning and 0.0001 for online reinforcement learning, mini-batch size of 256 samples, and reward discount factor $\gamma = 0.9$. The network has 2 hidden layers with 256 neurons each. These hyper-parameters (neural network structure, learning rate, mini-batch size, etc.) are chosen based on a few empirical training trials. We refer to one update of the neural network at the end of each time slot as one *step* in this section. It takes about half an hour to finish offline training in our setup.

Baselines. We compare DL^2 with the following baselines.

- Dominant Resource Fairness (DRF) [10]: It allocates resources to jobs based on the fairness of dominant resources. By default, we use DRF policy to guide supervised learning in DL^2, since it is widely adopted in existing cluster schedulers, e.g., YARN [9], Mesos [21].
- Tetris [48]: It preferentially allocates resources to jobs with the shortest remaining completion time and highest resource packing efficiency.
- Optimus [12]: It is a customized scheduler for DL workloads, which builds a performance model for deep learning jobs to estimate remaining training time and adopts a greedy heuristic to schedule jobs.
- OfflineRL [14], [16]: Several recent studies [14], [16] adopt offline RL method for single-task job scheduling or execution order scheduling. We adapt their methods to compare with them. Our offline reinforcement learning algorithm adopts same training techniques (e.g., experience replay, actor-critic) as [14], [16], and also uses the training data generated by an analytical performance model [12] in a simulation environment.

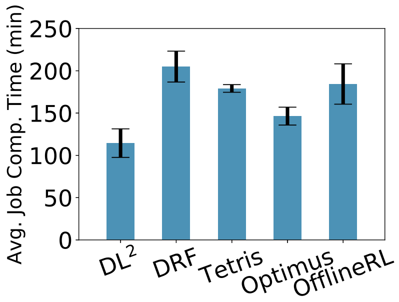


Fig. 9. Performance comparison.

Wherever appropriate, we use separate training dataset and validation dataset. Both include job sequences generated using the job arrival and duration distributions from the trace. The random seeds are different when generating the datasets, to ensure that they are different.

6.3 Performance

We first compare the performance of DL^2 with baselines and show the overhead of dynamic scaling in our testbed.

Comparison. Fig. 9 shows that DL^2 improves average job completion time by 44.1 percent when compared to DRF. Tetris performs better than DRF but worse than DL^2: once it selects a job with the highest score in terms of resource packing and remaining completion time, it always adds tasks to the job until the number of tasks reaches a user-defined threshold. When compared to Optimus, DL^2 achieves 17.5 percent higher performance, since Optimus' estimation of training speed is inaccurate due to cluster interference and evolved MXNet framework (e.g., communication does not overlap with backward computation in Optimus' model). DL^2 also outperforms OfflineRL by 37.9 percent due to its online training using realistic feedback.

For a better understanding of DL^2's performance gain, Fig. 10 shows how the validated performance keeps improving during the training process, when the policy NN is trained using offline supervised learning only (green curve), online RL only (cyan curve), and offline supervised learning followed by online RL (green+blue). The average job completion time shown at each time slot (i.e., step) is computed over job sequence in the validation dataset, using

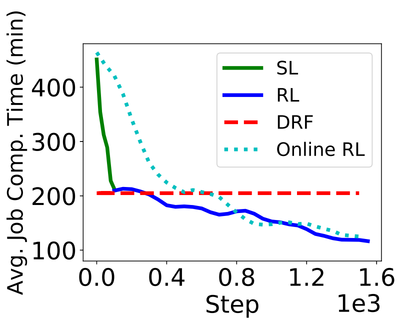


Fig. 10. Training progress comparison.

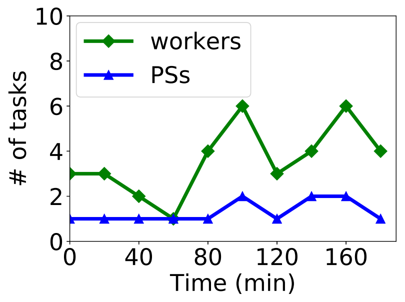


Fig. 11. Resource allocation of a Seq2Seq job.

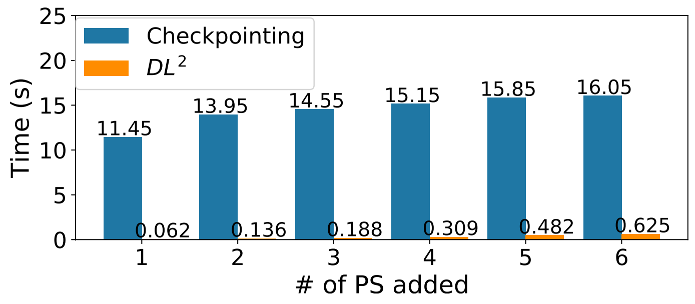


Fig. 13. Scaling overhead comparison.

the policy network trained (on the training dataset) at the current step. We see that with pure online RL, it takes hundreds of steps to achieve the same performance of DRF; with offline supervised learning, the performance quickly converges to the point that is close to DRF's performance within tens of steps (i.e., model updates); as we continue training the NN using online RL, the performance further improves a lot. The performance of DRF is fixed as its strategy does not change over time. Besides smaller average job completion time, we also observe that DL^2 has smaller cluster makespan (i.e., the total time elapsed from the arrival of the first job to the completion of all jobs), which indicates that DL^2 has higher resource efficiency [48]. For example, compared to DRF, DL^2 has 16 percent higher average GPU utilization (increased from 62 to 78 percent).

To understand DL^2's superior performance intuitively, we examine its decisions of resource allocation. Fig. 11 further shows how the numbers of PSs and workers of a Seq2Seq [24] job are adjusted by DL^2 during training. It shows that DL^2 tries to allocate twice the number of workers than the number of PSs, a good resource configuration for Seq2Seq (Fig. 2 in Section 2.1). We see that DL^2 is aware of the characteristics of a specific workload and learns a good policy for scheduling it.

Scaling Overhead. Fig. 13 compares the average training suspension time among all workers when checkpointing and our scaling approach is used respectively, when different numbers of PSs are added to a ResNet-50 [22] job. The training suspension duration at a worker in DL^2 is from when all the received iteration counts from PSs becomes equal to the scaling clock the worker received from the coordinator, to when the worker resumes training. The checkpoint-based approach takes tens of seconds due to model saving, container relaunching, and initialization before restarting training. The overhead in DL^2 is very small (i.e., tens of milliseconds), even if the time increases linearly with the number of PSs (since we add PSs one by one). We observe similar overhead when removing PSs.

We examine detailed time cost for the 4 steps during the scaling process (Section 5) for adding a PS when training different models. In Fig. 14, the models are listed in increasing order of their model sizes. We observe that the scaling process spends most time in step 3 and step 4, while the time for step 1 and step 2 is negligible. The larger a model is, the more time is spent on parameter movement (step 3). Note that only step 4 blocks worker training and is considered as overhead when compared to checkpointing. Step 3 and step 4 may happen concurrently.

We also measured the overall resource adjustment overhead. Similar to Optimus [12], we define the overhead as the percentage of time spent on adjusting resources for training jobs. In our experiments, the overall overhead is 0.4 percent, as compared to 2.54 percent in Optimus. The overhead is much less because DL^2 adopts elastic scaling instead of checkpointing to adjust job resources.

In the following, we carry out controlled large-scale simulations to examine various aspects of DL^2 design.

6.4 Generality

Job Arrival Patterns. We first investigate how job arrival patterns affect job completion time under two other job arrival processes: a uniform random process and a Poisson process. Fig. 12 shows the result when compared to Optimus. We see that DL^2 outperforms Optimus by about 20 percent in terms of average job completion time.

Training Completion Time Variation. To see how DL^2 handles practical performance variation (which white-box schedulers may not handle well), we vary the training speeds in each type of jobs to simulate variation in the training completion time of the same type of jobs (the total numbers of epochs to train remain the same). In Fig. 15, the variation indicates how the training speed deviates from the average speed (which can be faster or slower by the respective percentage).

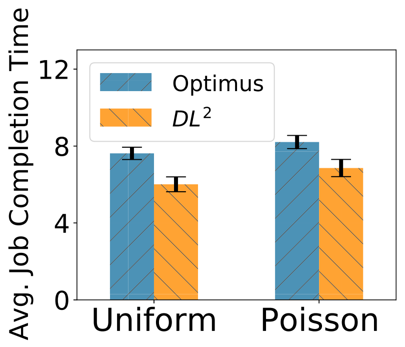


Fig. 12. Different arrival patterns.

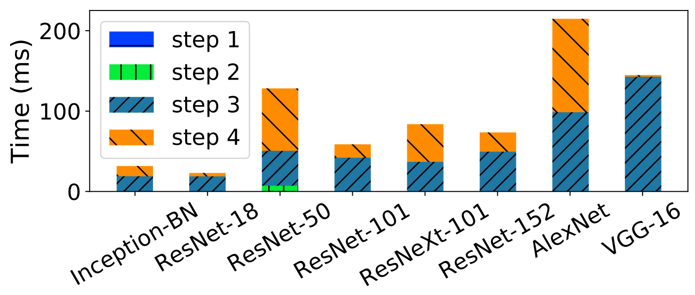


Fig. 14. Timing of scaling steps.

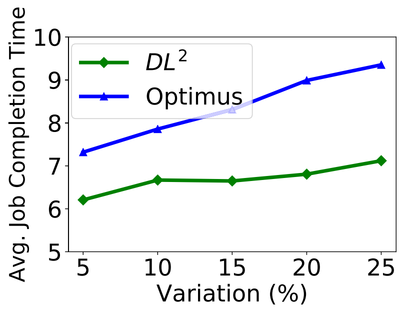


Fig. 15. Training completion time variation.

The average job completion time shown in all simulation figures is in time slots. We see that Optimus is more sensitive to the variation, as it can be easily stuck in local optimum: its scheduling relies on the convexity of the performance model, but training speed variation often breaks convexity. A further look at the completion time of each model (Table 1) reveals that Optimus has similar performance as DL^2 for small models (e.g., CTC and DSSM) but performs worse than DL^2 when the scale of job is large. This is due to the inaccurate performance estimation in Optimus when the numbers of PSs and workers are large.

Total Training Epoch Estimation. DL^2 uses the total number of training epochs of jobs as input, estimated by users. The estimated total number of epochs may well be different from the actual numbers of epochs the jobs need to train to achieve model convergence. We examine how DL^2 performs under different estimation errors: suppose v epochs is fed into DL^2 as the total epoch number that a job is to train, but $v \cdot (1 + error)$ or $v \cdot (1 - error)$ is the actual number of trained epochs for the job's training convergence. Fig. 16 shows that the average job completion time increases slightly when the estimation error is larger. It still outperforms DRF (which is oblivious of the estimation errors) by 28 percent when the error is 20 percent.

Unseen Job Types. We investigate whether DL^2 can adapt to jobs training new models. We train the neural network using the first four categories of models (Table 1) in the supervised learning phase and the first 1000 steps of the online RL phase. At step 1000 and step 2000 of the RL phase (i.e., the red dots in Fig. 17), we submit jobs training two new categories of models. In the case of the "ideal" baseline, we train the NN using all categories of jobs in Table 1 from the beginning. Fig. 17 shows the average job completion time achieved using the trained NN at each time respectively, for decision making over the validation dataset. DL^2 gradually achieves the same performance as the "ideal" baseline, showing its capability to handle new types of DL jobs coming on the go.

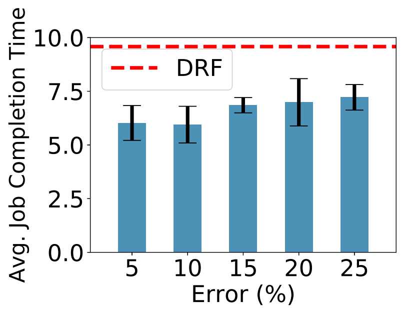


Fig. 16. Performance under different epoch estimation errors.

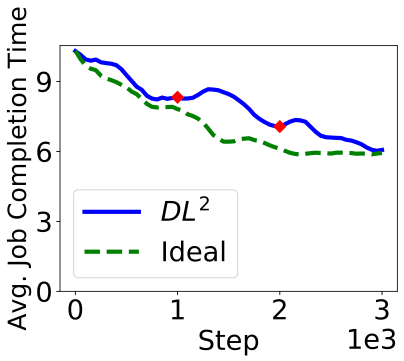


Fig. 17. Handling new types of jobs.

Other Scheduling Strategies for Supervised Learning. We change the default DRF used in supervised learning of DL^2 to two other heuristics, First-In-First-Out (FIFO) and Shortest-Remaining-Time-First (SRTF). Fig. 18 shows average job performance when DL^2 uses each of these strategies in its supervised learning phase, when the NN trained only using supervised learning, or using both supervised learning and online RL, is evaluated on the validation dataset. In both cases, the performance is significantly improved with DL^2, beyond what the existing scheduling strategy in the cluster can achieve (41.3 percent speedup in the case of SRTF).

6.5 Deep Dive

6.5.1 *Neural Network Architecture*

Concurrent Job Number. We investigate how the maximal number of concurrent jobs to schedule in a time slot, J specified in the NN input, affects the performance of DL^2 when applying the trained NN (after supervised learning and reinforcement learning) on the validation dataset. The maximal number of uncompleted jobs in all time slots is around 40; when the concurrent job number is larger than J, we schedule the jobs in batches of J jobs, according to their arrival sequence. In Fig. 19a, we observe that the performance suffers when J is small, possibly because the NN is not trained on a global view when jobs are fed into the NN in batches in each time slot. Setting J to be large enough to accommodate the maximal number of concurrent jobs gives better results.

Number of Neurons. We fix the number of hidden layers in DL^2's NN to 1 and vary the number of neurons in the

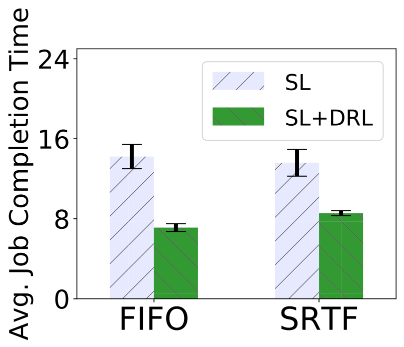
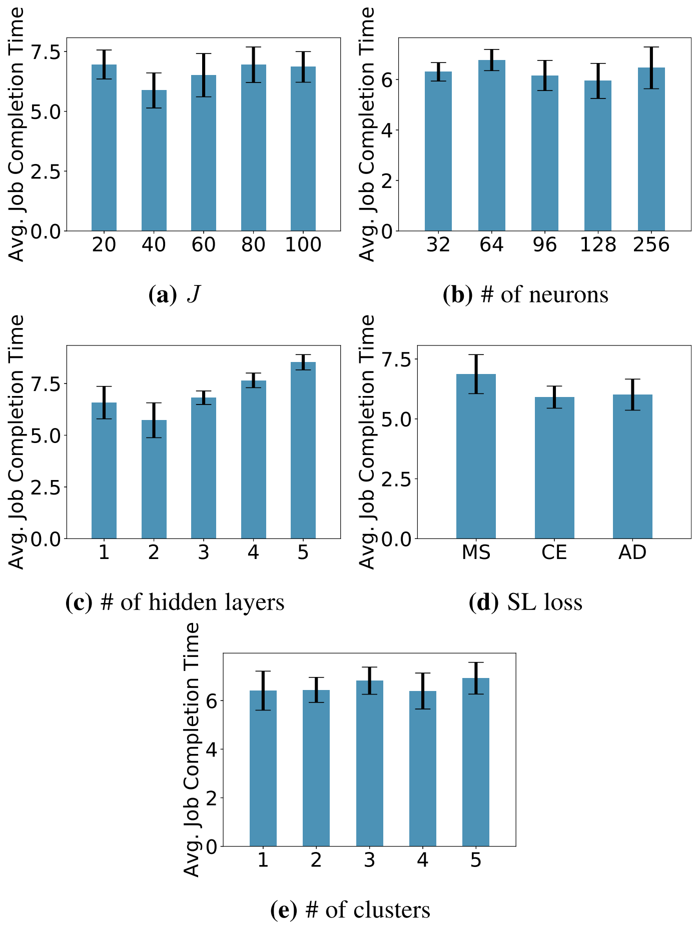


Fig. 18. Different existing scheduling strategies.



(a) J **(b)** # of neurons **(c)** # of hidden layers **(d)** SL loss **(e)** # of clusters

Fig. 19. Deep dive.

hidden layer. Fig. 19b shows that the best performance is achieved when there are 128 neurons. The performance degrades once the number exceeds 256. Since the size of the input layer is 180, it shows that the number of neurons should be a bit smaller than the input size to achieve good performance.

Number of Hidden Layers. Next, we fix the number of neurons to 128 and vary the number of hidden layers. As shown in Fig. 19c, the neural network of 2 hidden layers yields the best performance and minimal variance. Note that we use a fixed number of neurons for each layer and the learning rate is fixed. Further tuning these hyper-parameters for deeper networks may improve performance, though it generally takes longer training time.

Bundle Action. We remove the bundle action, i.e., allocating a worker/PS pair to a job, from our network, and rerun the training. As shown in Table 2, the job completion time is 35.1 percent worse than the case with bundle actions (row 3 in Table 2). Introducing the bundle actions can reduce the number of decisions (inferences) made in a time slot in general, which enables more efficient action space exploration.

6.5.2 Training Design

SL Loss Function. We evaluate three common loss functions for supervised learning, i.e., Mean Square, Cross Entropy (the default) and Absolute Difference [49] and use them to retrain our scheduler, respectively. We observe similar performance with these loss functions, while adopting Cross Entropy achieves the best performance (Fig. 19e). This is because Mean Square or Absolute Difference emphasize incorrect or suboptimal output, while only the correct or optimal output contributes to the loss when using Cross Entropy.

TABLE 2
Effectiveness of Training Techniques

Without	Avg. Job Completion Time	Slowdown (%)
-	5.724 ± 0.844	0
Bundle action	7.734 ± 0.339	35.1
Actor-critic	6.929 ± 0.477	21.1
Exploration	7.372 ± 0.548	28.8
Experience replay	7.988 ± 0.102	39.6

Reward Function. We evaluate another reward function with DL^2, which sets the reward of each action (that adds some worker/PS to a job) as the normalized number of epochs trained by that job in the time slot. We find that its performance is 29.1 percent worse. Our default reward function considers all jobs' progress, enabling the policy network to learn to schedule from a global perspective.

Actor-Critic. To see how the actor-critic algorithm affects training, we remove the value network but only train the policy network. As widely adopted in RL community, we use the exponential moving average of rewards as a baseline in place of the output of the value network in gradient computation of the policy network. As shown in Table 2, with the value network, the performance is 21.1 percent better. 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.

Job-Aware Exploration. We examine how exploration contributes to the performance. We find that without exploration, the performance is 28.8 percent worse (Table 2), as online RL is stuck in a local optimal policy.

Experience Replay. To examine the effectiveness of our prioritized experience replay, We disable experience replay and see how performance changes. Table 2 shows that the average job completion time is degraded by 39.6 percent, indicating that experience replay is critical for training.

Federated Training. Federated training enables multiple clusters to learn a global DL^2 model collaboratively. We study how the number of clusters affects the policy training, by implementing the A3C [37] algorithm, which trains a global policy NN using multiple DL^2 schedulers with different training datasets, each for one cluster. Fig. 19e shows that the global performance remains stable when we increase the number of clusters. We have also observed that with more clusters, the policy NN converges much faster due to the use of more training datasets: if there are x clusters, the NN converges almost x times faster. The preliminary result also suggests the possibility of dividing a single massive cluster into loosely coupled sub-clusters where each runs a DL^2 scheduler for resource allocation, if scalability issue arises.

7 Discussion and Future Directions

More Scheduling Features. Besides average job completion time, DL^2 can implement other scheduling features by adjusting the learning objective. For example, we can incorporate resource fairness by adding a quantified fairness term in the reward function.

All-Reduce Architecture. All-reduce architecture [50], where workers train model replicas and exchange updated model parameters directly with each other, is supported in Caffe2 [41], CNTK [51], etc. In this paper, we consider all-reduce as a special PS architecture without parameter servers, and hence we do not need to build another policy network for all-reduce jobs. Instead, we use the same policy neural network to schedule PS and all-reduce jobs, with the neural network trained using both PS and all-reduce traces. Specifically, when training the policy network using all-reduce traces, the input states of parameter servers (e.g., the number of allocated parameter servers $\vec{u}$) are padded with zeros and the output actions of parameter servers are masked.

Job Placement. While we use the default placement policy in this work, the placement of workers and PSs can potentially be decided by RL too. Using one NN to produce both resource allocation and placement decisions is challenging, mainly because of the significantly larger action space. RL using a hierarchical NN model [52] might be useful in making resource allocation and placement decisions in a hierarchical fashion.

Practical Deployment. In practical deployment, the following two issues may need to be considered: (1) adversarial attacks that fool a neural network with malicious input; (2) neural network monitoring that detects exceptional scheduling. These are interesting directions to explore, with progress in security research and more in-depth understanding of neural networks.

AutoML for Better Performance. We adopted a simplified approach for setting hyper-parameters and neural network architecture in DL^2. Tuning hyper-parameters or network architecture to ensure the best performance is always a challenging and tedious task. Without careful hyper-parameter or network architecture tuning, DL^2 already achieves good performance; we seek automated machine learning (AutoML) methods [53] [54] to tune the neural network to further improve the performance of DL^2 in the future.

8 Related Work

Deep Reinforcement Learning in System Research. Applying deep reinforcement learning to resource allocation or job scheduling is not new. Many recent studies use DRL for resource allocation, device placement, video streaming and IoT. Mao *et al.* [14] and Chen *et al.* [28] use offline DRL for job scheduling in cloud clusters, to minimize average job slowdown. Their NNs select the jobs (single-task jobs) to run with static resource allocation. The NNs are trained offline: multiple job arrival sequences are used as training examples; each example is repeatedly trained for multiple epochs. Bao *et al.* [26] use DRL for job placement to minimize runtime interference. Adjustment of resources during job execution is not in the scope of the above studies. Mao *et al.* [15], [16] learn an NN to schedule acyclic dataflow graphs in Spark, in terms of parallelism level and execution order of dependent tasks in the jobs, using offline training. For distributed ML jobs that we consider, PS and worker tasks are started at the same time (hence no need of execution ordering among them) and the dependency is not acyclic among the tasks. Besides, Decima [16] slots resources into homogeneous pieces to allocate to tasks. This cannot be applied to DL jobs, as workers and PSs usually require different resource configurations. We adapt their offline RL methods to our scenario and show that offline RL methods are worse than DL^2 (Section 6).

Mirhoseini *et al.* [52], [55] use DRL to optimize placement of a computation graph, to minimize running time of an individual TensorFlow job. Xu *et al.* [56] use DRL to select routing paths between network nodes for traffic engineering. Mao *et al.* [57] dynamically decide video streaming rates in an adaptive streaming system with DRL. All these studies resort to offline RL training, using data generated by analytical models or simulators. In contrast, we use offline supervised learning to prepare our NN and then online RL with job-aware exploration to further improve the NN.

Cluster Scheduling. SLAQ [11] adopts online fitting to estimate the training loss of convex algorithms, for scheduling jobs training classical ML models. Dorm [58] uses a utilization-fairness optimizer to schedule ML jobs. These work do not focus on distributed DL jobs using the parameter server architecture. Optimus [12] proposes a dynamic resource scheduler based on online-fitted resource-performance models. Bao *et al.* [13] design an online scheduling algorithm for DL jobs. These studies rely on detailed modeling of DL jobs and simplified assumptions in their design, and can not adapt to workload changes or framework changes automatically. Gandiva [59] exploits intra-job predictability to time-slice GPUs efficiently across multiple jobs, and dynamically migrate jobs to better-fit GPUs. They do not consider resource allocation adjustment and resource allocation with GPU sharing will be an intriguing future direction to explore. Tiresias [60] schedules jobs with partial information to minimize job completion time, but the scheduler does not resize job resources during runtime. ByteScheduler [61] and P3 [62] schedule network communication in DL clusters by exploiting the communication patterns of distributed DL training. These projects are complementary to DL^2.

Instead of DRL, some classical studies adopt different feedback control methods (e.g., fuzzy control theory [63], genetic algorithm [64]) in cluster resource scheduling. For example, Chen *et al.* [65] divide user requirements and available resources into several fuzzy levels and process user fuzzy requirements to improve the QoS of cloud computing. Fuzzy logic theory requires human-designed fuzzy rules and expert knowledge on cluster workloads and resources. Mona *et al.* [66] propose a genetic algorithm based scheduler for computational grids. These methods are rarely applied in recent cluster schedulers (e.g., Yarn [9], Mesos [21], Kubernetes [67]). Compared to DL^2, they rely on heuristics or rules, which cannot automatically embed workload-specific characteristics or adapt to workload changes.

9 Conclusion

We present DL^2, a DL-driven scheduler for DL clusters, which expedites job completion globally with efficient resource utilization. DL^2 starts from offline supervised learning, to ensure basic scheduling performance comparable to the existing cluster scheduler, and then runs in the live DL cluster to make online scheduling decisions, while improving its policy through reinforcement learning using live feedback. Our testbed experiments and large-scale trace-driven simulation verify DL^2's low scaling overhead, generality in various scenarios and outperformance over hand-crafted heuristics.

ACKNOWLEDGMENTS

This work was supported by Alibaba Group through Alibaba Innovative Research (AIR) Program, and grants from Hong Kong RGC under the contracts HKU 17204619 and 17208920.

REFERENCES

[1] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. Int. Conf. Learn. Representations*, 2015.
[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Advances Neural Inf. Process. Syst.*, vol. 25, 2012, pp. 1097–1105.
[3] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2013, pp. 6645–6649.
[4] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 265–283.
[5] T. Chen *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," in *Proc. Neural Inf. Process. Syst. Workshop Mach. Learn. Syst.*, 2016.
[6] E. P. Xing *et al.*, "Petuum: A new platform for distributed machine learning on big data," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2015, pp. 1335–1344.
[7] "PaddlePaddle," 2019. [Online]. Available: http://www.paddlepaddle.org/
[8] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015. pp. 1–17.
[9] V. K. Vavilapalli *et al.*, "Apache hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp, 1–6.
[10] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 323–336.
[11] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "SLAQ: Quality-driven scheduling for distributed machine learning," in *Proc. Symp. Cloud Comput.*, 2017, pp. 390–404.
[12] 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.*, 2018, pp. 1–14.
[13] Y. Bao, Y. Peng, C. Wu, and Z. Li, "Online job scheduling in distributed machine learning clusters," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 495–503.
[14] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. 15th ACM Workshop Hot Topics Netw.*, 2016, pp. 50–56.
[15] H. Mao, M. Schwarzkopf, S. Venkatakrishnan, and M. Alizadeh, "Learning graph-based cluster scheduling algorithms," in *Proc. ACM Special Interest Group Data Commun.*, 2018, pp. 270–288.
[16] 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.*, 2019, pp. 270–288.
[17] "Kubernetes," 2019. [Online]. Available: https://kubernetes.io
[18] "Source code," 2021. [Online]. Available: https://github.com/pengyanghua/DL2
[19] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 583–598.
[20] J. Jiang, L. Yu, J. Jiang, Y. Liu, and B. Cui, "Angel: A new large-scale machine learning system," *Nat. Sci. Rev.*, vol. 5, pp. 216–236, 2017.
[21] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 295–308.
[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[23] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Representations*, 2015.
[24] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 1243–1252.
[25] L. Chen, J. Lingys, K. Chen, and F. Liu, "AuTO: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2018 pp. 191–205.
[26] Y. Bao, Y. Peng, and C. Wu, "Deep learning-based job placement in distributed machine learning clusters," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 505–513.
[27] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998.
[28] W. Chen, Y. Xu, and X. Wu, "Deep reinforcement learning for multi-resource multi-cluster job scheduling," in *Proc. IEEE Int. Conf. Netw. Protocols*, 2017. [Online]. Available: https://iqua.ece.toronto.edu/icnp17/program.html
[29] P. Goyal *et al.*, "Accurate, large minibatch SGD: Training imageNet in 1 hour," 2017, *arXiv: 1706.02677*.
[30] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chap. Assoc. Comput. Linguistics*, Hum. Lang. Technol., vol. 1, 2019, pp. 4171–4186.
[31] O. Vinyals *et al.*, "StarCraft II: A new challenge for reinforcement learning," 2017, *arXiv: 1708.04782*.
[32] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proc. 27th Int. Conf. Int. Conf. Mach. Learn.*, 2010, pp. 807–814.
[33] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.
[34] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. 12th Int. Conf. Neural Inf. Process. Syst.*, vol. 99, 2000, pp. 1057–1063.
[35] S. Mannor, D. Peleg, and R. Rubinstein, "The cross entropy method for classification," in *Proc. 22nd Int. Conf. Mach. Learn.*, 2005, pp. 561–568.
[36] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, pp. 229–256, 1992.
[37] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," in *Proc. 33rd Int. Conf. Int. Conf. Mach. Learn.*, 2016, pp. 1928–1937.
[38] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.
[39] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons, "Proteus: Agile ML elasticity through tiered reliability in dynamic resource markets," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 589–604.
[40] A. Qiao *et al.*, "Litz: Elastic framework for high-performance distributed machine learning," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 631–643.
[41] "Caffe2," 2019. [Online]. Available: https://caffe2.ai/
[42] Y. Shen, X. He, J. Gao, L. Deng, and G. Mesnil, "A latent semantic model with convolutional-pooling structure for information retrieval," in *Proc. 23rd ACM Int. Conf. Inf. Knowl. Manage.*, 2014, pp. 101–110.
[43] "HDFS," 2014. [Online]. Available: https://wiki.apache.org/hadoop/HDFS
[44] "Docker," 2019. [Online]. Available: https://www.docker.com/
[45] "MXNet official examples," 2019. [Online]. Available: https://github.com/apache/incubator-mxnet/tree/master/example
[46] "ImageNet dataset," 2019. [Online]. Available: http://www.image-net.org
[47] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Representations*, 2015.
[48] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 455–466.
[49] "TFLearn objectives," 2019. [Online]. Available: http://tflearn.org/objectives/

[50] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch, “Ako: Decentralised deep learning with partial gradient exchange,” in *Proc. 7th ACM Symp. Cloud Comput.*, 2016, pp. 84–97.

[51] “CNTK,” 2019. [Online]. Available: https://github.com/Microsoft/CNTK

[52] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, “A hierarchical model for device placement,” in *Proc. Int. Conf. Learn. Representations*, 2018.

[53] T. Swearingen, W. Drevo, B. Cyphers, A. Cuesta-Infante, A. Ross, and K. Veeramachaneni, “ATM: A distributed, collaborative, scalable system for automated machine learning,” in *Proc. IEEE Int. Conf. Big Data*, 2017, pp. 151–162.

[54] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, “Efficient and robust automated machine learning,” in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 2755–s2763.

[55] A. Mirhoseini *et al.*, “Device placement optimization with reinforcement learning,” in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 2430–2439.

[56] Z. Xu *et al.*, “Experience-driven networking: A deep reinforcement learning based approach,” in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 1871–1879.

[57] H. Mao, R. Netravali, and M. Alizadeh, “Neural adaptive video streaming with pensieve,” in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 197–210.

[58] 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.*, 2017, pp. 1–6.

[59] W. Xiao *et al.*, “Gandiva: Introspective cluster scheduling for deep learning,” in *Proc. 13th USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 595–610.

[60] J. Gu *et al.*, “Tiresias: A GPU cluster manager for distributed deep learning,” in *Proc. 16th USENIX Conf. Netw. Syst. Des. Implementation*, 2019, pp. 485–500.

[61] Y. Peng *et al.*, “A generic communication scheduler for distributed DNN training acceleration,” in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 16–29.

[62] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, “Priority-based parameter propagation for distributed DNN training,” in *Proc. Mach. Learn. Syst.*, vol. 1, 2019, pp. 132–145.

[63] “Fuzzy logic,” 2020. [Online]. Available: https://en.wikipedia.org/wiki/Fuzzy_logic

[64] “Genetic algorithm,” 2020. [Online]. Available: https://en.wikipedia.org/wiki/Genetic_algorithm

[65] C. Zhijia, Z. Yuanchang, D. Yanqiang, and S. Feng, “A dynamic resource scheduling method based on fuzzy control theory in cloud environment,” *J. Control Sci. Eng.*, vol. 2015, 2015, Art. no. 34.

[66] M. Aggarwal, R. D. Kent, and A. Ngom, “Genetic algorithm based scheduler for computational grids,” in *Proc. 19th IEEE Int. Symp. High Perform. Comput. Syst. Appl.*, 2005, pp. 209–215.

[67] Google, “Kubernetes,” 2019. [Online]. Available: https://kubernetes.io/

Yanghua Peng received the BEng degree from the Department of Computer Science and Technology, Wuhan University, China, in 2015, and the PhD 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.

Yixin Bao (Student Member, IEEE) received the BEng degree from the Department of Automation, Xi’an Jiaotong University, in 2015, and the PhD 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.

Yangrui Chen received the BE degree from the Department of Automation, Tsinghua University, China, in 2017. He is working toward the PhD degree with the Department of Computer Science, University of Hong Kong. His research interests include distributed machine learning and graph neural networks.

Chuan Wu (Senior Member, IEEE) received the PhD 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, University of Hong Kong, where she is currently a professor. Her current research interests include the areas of cloud computing, distributed machine learning systems, network function virtualization, and intelligent elderly care technologies.

Chen Meng received the BE degree in computer science and technology from Jilin University, in 2011, and the PhD degree from the Supercomputing Center of Chinese Academy of Sciences, in 2016. She joined NAOC as a postdoctoral research fellow in 2019. Her research interests include massive heterogeneous parallel computing in deep learning and science simulations.

Wei Lin is currently the senior director of PAI & chief architect of big-data computation platform at Alibaba. He has more than 15 years of experience specializing in backend/infrastructure, distributed system development, storage and a large scale computation system include batch, streaming and machine learning. He has published many papers in top computer system conferences, such as NSDI, SoCC, and OSDI.