Chic: Experience-driven Scheduling in Machine Learning Clusters

Yifan Gong[1], Baochun Li[1], Ben Liang[1], Zheng Zhan[2]
[1]Department of Electrical and Computer Engineering, University of Toronto
[2]College of Engineering and Computer Science, Syracuse University
ygong@ece.utoronto.ca,bli@ece.utoronto.ca,liang@ece.utoronto.ca,zzhan03@syr.edu

ABSTRACT

Large-scale machine learning (ML) models are routinely trained in a distributed fashion, due to their increasing complexity and data sizes. In a shared cluster handling multiple distributed learning workloads with a parameter server framework, it is important to determine the adequate number of concurrent workers and parameter servers for each ML workload over time, in order to minimize the average completion time and increase resource utilization. Existing schedulers for machine learning workloads involve meticulously designed heuristics. However, as the execution environment is highly complex and dynamic, it is challenging to construct an accurate model to make online decisions. In this paper, we design an experience-driven approach that learns to manage the cluster directly from experience rather than using a mathematical model. We propose *Chic*, a scheduler that is tailored for scheduling machine learning workloads in a cluster by leveraging deep reinforcement learning techniques. With our design of the state space, action space, and reward function, *Chic* trains a deep neural network with a modified version of the cross-entropy method to approximate the policy for assigning workers and parameter servers for future workloads based on the experience of the agent. Furthermore, a simplified version named *Chic-Pair* with a shorter training time for the policy is purposed by assigning workers and parameter servers in a pair. We compare *Chic* and *Chic-Pair* with state-of-the-art heuristics, and our results show that *Chic* and *Chic-Pair* are able to reduce the average training time significantly for machine learning workloads under a wide variety of conditions.

CCS CONCEPTS

• **Computing methodologies → Machine learning**; **Reinforcement learning**; • **Computer systems organization** → *Cloud computing*.

KEYWORDS

Distributed Machine Learning, Deep Reinforcement Learning, Workload Scheduling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IWQoS '19, June 24–25, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6778-3/19/06…$15.00
https://doi.org/10.1145/3326285.3329065

ACM Reference Format:
Yifan Gong[1], Baochun Li[1], Ben Liang[1], Zheng Zhan[2]. 2019. *Chic:* Experience-driven Scheduling in Machine Learning Clusters. In *IEEE/ACM International Symposium on Quality of Service (IWQoS '19), June 24–25, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3326285.3329065

1 INTRODUCTION

In recent years, modern machine learning (ML) techniques, such as deep neural networks (DNNs), have been successfully applied to solve practical problems in areas such as image classification and speech recognition. With accelerating growth in the volume of training data, the time and resources needed to train DNN models have increased substantially. It is therefore customary to train these models in a *distributed* fashion with a large number of *worker* nodes. Each worker node contains the entire model but only processes a portion of the training data, and *parameter server (PS)* nodes are used to facilitate the synchronization of gradients across these worker nodes [1].

In a shared ML cluster with multiple training workloads submitted over time, assigning worker nodes and PS nodes appropriately for different workloads is the key to expedite the training process while maximally leverage the resources. For existing ML clusters, Google uses Borg as the scheduler [2], while Microsoft, Tencent, and Baidu use YARN-like schedulers [3]. However, it is difficult for existing cluster schedulers to achieve high levels of efficiency in terms of scheduling ML workloads. First, these schedulers will assign a fixed amount of resources to a workload upon its arrival, and the allocation will remain the same during the training process unless the cluster operator manually reconfigures the resource composition, or a workload owner resubmits the workload as new. Therefore, scheduled workloads cannot benefit from available resources released by newly finished workloads during a long training process. Second, these schedulers are designed for general resource management and not tailored for ML workloads, leaving space for further improvement.

Considering the limitations of existing cluster schedulers, new schedulers tailored for ML clusters have recently been proposed, with the objectives of improving the average completion time of ML workloads and resource utilization. For example, Peng *et al.* proposed a practical resource scheduler for ML clusters called *Optimus* [4], which uses an online fitting method to predict the model convergence during training, and sets up a performance model to estimate the training speed as a function of allocated resources for each workload. Based on these models, the authors designed a simple heuristic to allocate resources for ML workloads. The key idea behind the heuristic is to iteratively add a worker node or a PS node to the workload to achieve the largest marginal reduction in the predicted completion time. The predicted completion time is based on the assumption that the schedule for the workload remains the same. However, since *Optimus* has to change the schedule from time to time to suit system dynamics, the heuristic might not be effective in the long run. In addition, since *Optimus* highly depends on the accuracy of the performance model, inaccuracies in the model may lead to heuristics that are far from optimal.

In this paper, we propose *Chic*, a new scheduling policy for scheduling ML workloads based on deep reinforcement learning (DRL). DRL [5] provides a promising technique for enabling effective experience-driven model-free control. Complex systems and decision-making policies could be modeled as DNNs by adopting reinforcement learning. By continuing to learn, the agent can optimize for a specific environment and work under varying conditions. There are several successful attempts in applying DRL to solve resource management problems. For example, Mao *et al.* employed the policy gradient method to compute the time for jobs to be scheduled in a cluster [6]. Mirhoseini *et al.* tried to apply DRL to expedite the training of ML workloads [7] by leveraging model parallelism. Both achieved better performance compared with handcrafted heuristics. We believe DRL is a promising and suitable solution to our problem, based on its features and previous successful examples.

Highlights of our original contributions in this paper are as follows. First, we model the state space, action space, and reward function in the case of scheduling ML workloads in a distributed ML cluster with the objective of minimizing the average completion time for a DRL agent. Second, a modified version of the cross-entropy method is proposed to train the policy network. Next, a simplified version of the scheduler is proposed to reduce the training time. Finally, we propose a new framework to implement our design, and conduct extensive simulations to show its effectiveness compared with the state-of-the-art ML workload scheduler.

The remainder of this paper is organized as follows. We state the problem of scheduling ML workloads in a distributed cluster and point out the importance and difficulty in solving this problem in Section 2. Preliminaries about DRL are introduced in Section 3. A concrete design of *Chic*, including the framework, the components of the DRL agent, and the training algorithm, is presented in Section 4. Our implementation details and simulation results are presented in Sections 5 and 6, respectively. Finally, we differentiate our work from related research efforts in Section 7 and conclude the paper in Section 8.

2 PRELIMINARIES AND PROBLEM STATEMENT

In this section, we first briefly present some preliminaries on distributed learning, we then present the statement of the problem and the motivations for the design of *Chic*.

2.1 Distributed Learning

Distributed learning allows us to leverage multiple machines to expedite the training process. Various ML frameworks, e.g., TensorFlow [8] and MXNet [9], support the paradigm of distributed learning. There are two ways to distribute the workload of training a neural network across multiple worker nodes. The first is data parallelism, in which the dataset is split among worker nodes. Each worker node has a complete copy of the model and learns the parameters only on a subset of the dataset. The alternative to data parallelism is model parallelism: rather than partitioning the dataset, the model is split across worker nodes. Each worker node trains a part of the model across the entire dataset. Data parallelism is more widely used for two reasons. First, it is easier and more efficient to partition the dataset than to split the model. Second, the memory of modern GPUs is able to store the entirety of most ML models, eliminating the need for partitioning the model. In this paper, we focus on ML workloads adopting data parallelism.

With data parallelism, each worker node only computes parameters based on a portion of the dataset. Hence, it is necessary to combine the parameters computed by each worker node, which is traditionally achieved by using a parameter server (PS) framework [1]. With a PS framework, each workload needs one or more worker nodes and PS nodes to meet the training requirement. Worker nodes are responsible for the training on the dataset and PS nodes are used to facilitate parameter exchanges. In the case of multiple PS nodes, the parameters within an ML model are divided equally and each PS node hosts one partition of the parameters. The dataset of an ML workload is stored in a distributed storage system (e.g., HDFS). To assign data partitions to multiple workers, a master node will divide the dataset to equal-sized data trunks. Each worker will fetch a data chunk upon its start and compute the gradients with the data trunk. Each data trunk is further divided into mini-batches. Once the worker processes one mini-batch, it will synchronize the parameters with other worker nodes by communicating the gradients to the PS nodes. Then, the PS node will update parameters using a formula, such as new_parameter = old_parameter − leaning_rate · gradient. After that, the worker node has to obtain the newly updated parameters from the PS nodes and begin processing the next mini-batch. The time interval between processing mini-batches is an iteration. An epoch is a duration when the whole dataset is trained once. An ML workload typically requires several epochs, which is equal to multiple iterations in total, to achieve training requirements.

Depending on whether the training progress is synchronized among worker nodes, the training mode could be classified into synchronous training and asynchronous training. For synchronous training, PS nodes update the parameters after they have collected the gradients from all worker nodes. Therefore, the amount of data used for parameter update in each iteration, termed batch size, is equal to the size of the mini-batch multiplied by the number of workers. In asynchronous training, the PS nodes update the parameters as soon as they receive the gradients from any of the worker nodes, and use updated parameters to respond to any future pulls. Synchronous training is more widely adopted as it typically requires fewer epochs to converge and is more stable compared with asynchronous training. Therefore, synchronous training is considered within this paper.

2.2 Problem Statement

As building a computing cluster is prohibitively expensive, a cluster is usually shared by many users. We represent the resource capacity of a cluster with K types of resources as $C = \{c_1, \cdots, c_K\}$, where c_k is the amount of type-k resource. A set of ML workloads with different training models and large datasets using synchronous

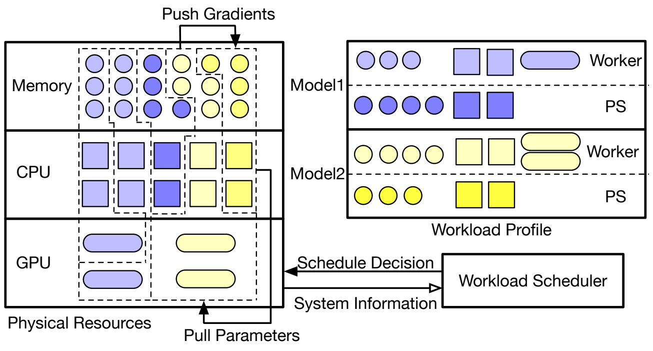


Figure 1: The distributed ML cluster.

training mode join the cluster in an online fashion. We characterize the information of job i as $s_i = \{s_i^m, s_i^B, s_i^w, s_i^p\}$, which includes the training model s_i^m, the remaining amount of iterations s_i^B to be trained, and the current allocation of workers s_i^w and PSs s_i^p for ML workload i. Each worker requires w_{ik} amount of type-k resources, and each PS requires p_{ik} amount of type-k resources for ML workload i, which are determined by its training model s_i^m.

Most ML workloads nowadays need a long time to converge even with distributed training because of the continuously growing model complexity and dataset size. For data parallel ML workloads with a PS framework, the number of worker nodes and PS nodes will greatly influence the completion time. With multiple workloads competing for a limited amount of physical resources, it is a significant problem how to manage the cluster and schedule the ML workloads to expedite the training process by efficiently utilizing the physical resources. A scheduler is needed in the cluster to determine the schedule, which is the allocation of worker nodes and PS nodes for the ML workloads currently in the system, to minimize the average completion time.

Assume there are I workloads in the cluster, a schedule $\{\{s_1^w, s_1^p\}, \cdots, \{s_I^w, s_I^p\}\}$ indicates the number of worker nodes and PS nodes assigned to every workload $i \in [I]$. The schedule should not exceed the physical resource capacity, which could be represented as $\sum_{i\in[I]}(s_i^w w_{ik} + s_i^p p_{ik}) \leq c_k, \forall k \in [K]$. Furthermore, A reasonable scheduler should change the schedule from time to time to suit the system dynamics caused by newly coming and finished workloads. Therefore in our system, a schedule decision will be kept only until a workload finishes, or a new workload arrives, which requires for a new schedule decision.

Fig. 1 illustrates a simple example of a distributed ML cluster. The cluster possesses three types of resources, and the workload scheduler makes a scheduling decision, allocating 2 workers and 1 PS for the first workload while 1 worker and 1 PS for the second workload. For each ML workload, the worker nodes will carry out the computation based on the dataset assigned to them and push the computed gradients to the PS nodes, wait for the PS nodes updating the parameters, and then pull newly updated parameters from the PS nodes within each iteration.

However, solving the scheduling problem in distributed ML cluster is challenging. More workers and PSs do not necessarily lead to shorter training time [4] for a single ML job. It is harder to decide the number of workers and PSs for multiple workloads so that the average completion time is minimized. Furthermore, since ML workloads join the cluster in an online fashion, the problem could not be solved with an offline integer linear programming model. Though previous approaches try to solve this problem by human-designed heuristics [4][10], they highly rely on the authors' understanding of the underlying system, which is hard to model accurately. The inaccuracy may lead the solution to be far from optimum.

DRL is a promising method to solve the online scheduling problem in ML clusters. Complex systems and decision-making policies could be modeled by neural networks, reducing the need for meticulously designed heuristics and carefully built performance models. In addition, DRL is able to deal with highly dynamic time-varying environments, which is precisely the case of the online scheduling problem. To the best of knowledge, we are the first to take advantage of the emerging DRL technique to enable experience-driven and model-free job scheduling in distributed ML clusters.

3 DEEP REINFORCEMENT LEARNING

In this section, we provide necessary background about DRL, which applies a DNN as function approximator to reinforcement learning (RL).

In the standard RL setting, an agent interacts with an environment E in an episodic case over discrete time steps [11]. At each time step t, the agent observes some state s_t, and is required to take an action a_t from a set of possible actions $\mathcal{A}$. Following the taken action, the state of the environment transits to s_{t+1} and the agent receives a reward r_t. The process continues until the agent reaches a terminal state after which the process restarts and a new episode begins. The states, actions, and rewards the agent experienced during one episode form a trajectory $x = (s_1, a_1, r_1, s_2 \cdots, s_T, a_T, r_T)$, where T is the last time step in the episode. The cumulative reward $R(x) = \sum_{t\in[T]} r_t$ measures how good the trajectory is by summing the rewards received at each time step. The agent's behavior is defined by a policy $\pi(a_t = a|s_t = s)$, mapping state s to a probability distribution over all actions $a \in \mathcal{A}$. In policy-based model-free RL methods, the policy is usually parameterized with parameters θ. For the problem of scheduling ML jobs in a cluster, an optimal policy $\pi(a|s;\theta^*)$ with parameters θ^* is the scheduling strategy we hope to obtain.

DNNs have been used as function approximators to solve large scale RL-tasks successfully in recent years [5][12]. One advantage of DNNs is that they do not need handcrafted features. In our paper, we use a DNN as the function approximator for the policy. To obtain the parameters θ in the policy DNN, a basic but efficient cross-entropy method could be utilized [13]. The objective is to maximize the reward $R(x)$ received by a trajectory x from an arbitrary set of trajectories $\mathcal{X}$. Suppose x^* is the corresponding trajectory at which the maximal cumulative reward is attained, and let ξ^* denote the maximum cumulative reward. We thus have: $R(x^*) = \xi^* = \max_{x\in\mathcal{X}} R(x)$.

Assume x has some probability density $f(x;u)$ with parameters u on $\mathcal{X}$. The estimation of the probability that the cumulative reward of a trajectory is greater than a fixed level ξ is $l = \mathbb{P}(R(x) \geq$

$\xi) = \mathbb{E}[\mathbf{1}_{\{R(x)\geq\xi\}}]$, where $\mathbf{1}_{\{R(x)\geq\xi\}}$ is the indicator function, that is, $\mathbf{1}_{\{R(x)\geq\xi\}} = 1$ if $R(x) \geq \xi$, and 0 otherwise. If ξ is chosen close to the unknown ξ^*, then l is a rare-event probability, which requires a large number of samples to estimate the expectation accurately. A better way is to use importance sampling. Let $f(x;v)$ be another probability with parameters v such that $f(x;v) = 0$ implies $\mathbf{1}_{\{R(x)\geq\xi\}} f(x;u) = 0$. Using the density $f(x;v)$, l could be represented as

$$l = \int \mathbf{1}_{\{R(x)\geq\xi\}} \frac{f(x;u)}{f(x;v)} f(x;v)\mathrm{d}x. \quad (1)$$

The optimal importance sampling probability for a fixed level ξ is given by $f(x;v^*) \propto |\mathbf{1}_{\{R(x)\geq\xi\}}| f(x;u)$, which is in general difficult to obtain. The idea of the cross-entropy method is to choose the importance sampling probability density $f(x;v)$ in a specified class of densities such that the distance between the optimal importance sampling density $f(x;v^*)$ and $f(x;v)$ is minimal. The distance $D(f_1, f_2)$ between two probability densities f_1 and f_2 is measured by the cross-entropy as $-\mathbb{E}_{x\sim f_1(x)}[\log f_2(x)]$. Hence, the optimal parameters could be obtained by the maximization problem $\max_v \int \mathbf{1}_{\{R(x)\geq\xi\}} f(x;u)\log f(x;v)\mathrm{d}x$, which can be estimated via sampling by solving with respect to parameters v the stochastic counterpart program:

$$\hat{v} = \arg\max_{v} \frac{1}{N} \sum_{n\in[N]} \mathbf{1}_{\{R(x_n)\geq\xi\}} \frac{f(x_n;u)}{f(x_n;w)} \log f(x_n;v), \quad (2)$$

where $x_1, \cdots, x_N$ are random samples from $f(x;w)$ for any reference parameter w. Initially, the parameters $u = \hat{v}_0$ are set randomly. By sampling with current importance sampling distribution in each iteration k, we create a sequence of levels $\hat{\xi}_1, \hat{\xi}_2, \cdots$ and the corresponding sequence of parameter vector $\hat{v}_0, \hat{v}_1, \cdots$. The sequence converges to the optimal performance ξ^* and the sequence $\hat{v}_0, \hat{v}_1, \cdots$ converges to the optimal parameter vector. Note that $\hat{\xi}_k$ is typically chosen as the $(1-\rho)$-quantile of the performance of the sampled trajectories. Sampling from an importance sampling distribution close to the theoretically optimal importance sampling density will produce optimal or near-optimal trajectories x^*. Often a smoothed updating rule with a smoothing parameter α is used, so that the parameter vector $\tilde{v}_k$ within the importance sampling density $f(x;v)$ after k-th iteration is taken as $\tilde{v}_k = \alpha\hat{v}_k + (1-\alpha)\tilde{v}_{k-1}$.

The probability of a trajectory $x \in \mathcal{X}$ is determined by the transition dynamics $p(s_{t+1}|s_t, a_t)$ of the environment and the policy $\pi(a_t|s_t;\theta)$. As the transition dynamics is determined by the environment and cannot be changed, the parameters θ in policy $\pi(a_t|s_t;\theta)$ are to be updated to improve the importance sampling density $f(x;v)$ of a trajectory x with high $R(x)$. Therefore, the parameter estimate at iteration k could be represented as

$$\hat{\theta}_k = \arg\max_{\theta_k} \sum_{n\in[N]} \mathbf{1}_{\{R(x_n)\geq\xi_k\}} \Big(\sum_{a_t, s_t \in x_n} \pi(a_t|s_t;\theta_k)\Big), \quad (3)$$

where $x_1, \cdots, x_N$ are sampled from $\pi(a|s;\tilde{\theta}_{k-1})$, $\tilde{\theta}_k = \alpha\hat{\theta}_k + (1-\alpha)\tilde{\theta}_{k-1}$. This equation suggests maximizing the likelihood of actions in trajectories with a high cumulative reward.

4 DESIGN

In this section, we present our design of *Chic*. We begin by giving an overview of the *Chic* architecture. Then we describe how to represent the online scheduling problem in a distributed ML cluster, as an RL task in terms of state space, action space, and reward. Finally, we present the training algorithm of *Chic*.

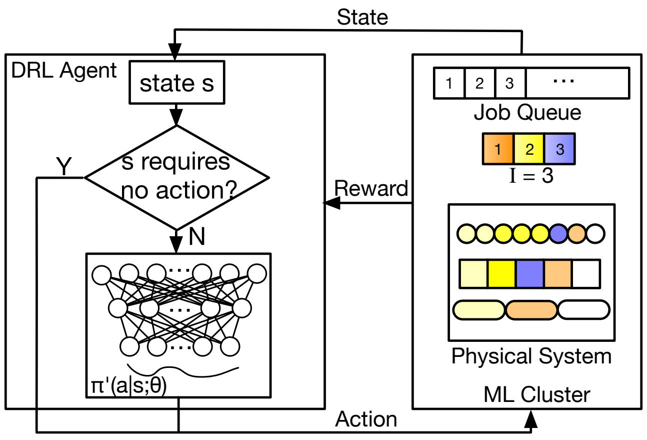


Figure 2: The architecture of *Chic*.

4.1 Architecture

We first give an overview of *Chic*, which is illustrated in Fig. 2. The key idea is the usage of a single DRL agent performing scheduling decisions for workloads within the ML cluster. The most important component of the DRL agent is the policy $\pi'(a|s;\theta)$, providing the probability distribution over all actions given a state s. The parameters θ in $\pi'(a|s;\theta)$ are learned from experiences collected during the interacting with the environment E, in a way similar to what is discussed in Section 3.

4.2 DRL formulation

No matter which specific DRL method is applied, we need to first design the state space, action space, and reward function. This design is the most critical step for the successful application of DRL to a practical problem. A well designed DRL agent should capture the key components of the problem without redundant or useless information.

State Space. To describe the system condition correctly for the DRL agent, the state should include the information of the ML workloads within the cluster. Let I_t be the total number of ML workloads at time step t. The DRL agent will focus on the scheduling of the first I workloads in the cluster, where I is selected to balance the tradeoff between learning performance and computational complexity. The detailed information $s_{it} = \{s^m_{it}, s^B_{it}, s^w_{it}, s^p_{it}\}$ for each ML workload $i \in [I]$ at the beginning of time step t is revealed to the agent. The workloads beyond the first I wait in the queue, and their number is summarized in the backlog component of the state, which is represented as N_t. Therefore, the state of the system at time step t could be represented as $s_t = \{s_{1t}, \cdots, s_{It}, N_t\}$. It contains the detailed information for each of the first I ML workloads and the number of ML workloads that are waiting in the queue. When there are $I_t \leq I$ workloads in the system, s_t still contains the information of I workloads by setting s^B_{it}, s^w_{it} and s^p_{it} as 0 for $i > I_t$ and keeping N_t as 0, until new ML workloads join the system.

Action Space. The action a_t is designed as the increase of a worker node or a PS node to a workload i at time step t. The action space for a system scheduling for the first I workloads is therefore given by $\mathcal{A} = \{1, \cdots, 2I, \emptyset\}$, so that $a_t = 2i-1$ refers to allocating one more worker node to workload i, while $a_t = 2i$ refers to increasing the allocation of PS node by one to workload i for $i \in [I]$. Furthermore, $a_t = \emptyset$ indicates the DRL agent does not wish to increase the allocation of worker nodes and PS nodes to any of the I workloads. We emphasize here that, though it may seem slow to change one worker or one PS per time step, the actual time to complete the schedule, i.e., until the physical resources are fully utilized, is short. Further details on this are discussed in Section 4.2.

Scheduling Process. *Chic* schedules ML workloads with two iterative phases. Initially, each workload has 0 worker node and 0 PS node. In the subsequent time steps, the DRL agent tries to allocate a worker or a PS to one workload $i \in [I]$ in each time step until either of the two conditions below is achieved:

(1) Resources in the cluster are fully utilized;
(2) The per-iteration running time of workload i will not decrease even if a worker node or a PS node is added to any workload $i \in [I]$.

Condition (1) means that no more workers or PSs for any workload $i \in [I]$ could be launched with the remaining resources in the system, and it could be formulated as

$$\min_i\{w_{ik}, p_{ik}\} + \sum_{i\in[I]}(s^w_{it}w_{ik} + s^p_{it}p_{ik}) > c_k, \exists k \in [K]. \tag{4}$$

Denoting by $g_i(s^w_{it}, s^p_{it})$, the per-iteration running time for a workload i with s^w_{it} amount of worker nodes and s^p_{it} amount of PS nodes as $g_i(s^w_{it}, s^p_{it})$, Condition (2) could be expressed as

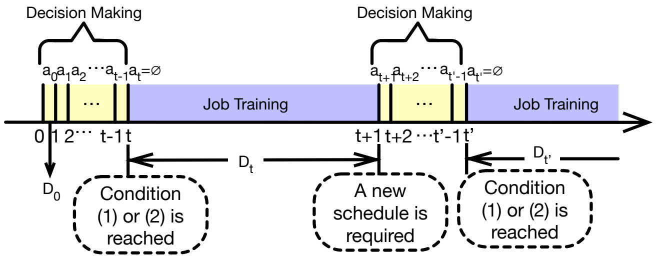
$$\min_i\{g_i(s^w_{it}+1, s^p_{it}), g_i(s^w_{it}, s^p_{it}+1)\} \geq g_i(s^w_{it}, s^p_{it}), \forall i \in [I]. \tag{5}$$

Reaching any of the above mentioned two conditions will yield a schedule for the I workloads. The time interval between having 0 worker node and 0 PS node for every workload $i \in [I]$ and obtaining a schedule is termed the *decision making phase*.

Upon obtaining the schedule, the system enters the job training phase and the I workloads begin training with the allocated worker nodes and PS nodes, till a new schedule is required. Different from *Optimus*, which demands a new schedule based on fixed time intervals, *Chic* requires a new schedule when either of the two situations below occur:

(1) A workload $i \in [I]$ is finished;
(2) A new workload arrives when there are fewer than I workloads in the system.

The occurrence of either of the situations indicates the resources in the cluster could be rescheduled to expedite the training of the workloads currently in the system. Therefore, by changing the schedule only when it is necessary, *Chic* ensures the efficient use of resources while avoiding the extra cost caused by unnecessary changes. When a new schedule is required, the worker node and PS node assigned to each workload $i \in [I]$ are both reset as 0, after which the decision making phase begins again.



Decision Making
Decision Making
$a_0 a_1 a_2 \cdots a_{t-1} a_t=\emptyset$
$a_{t+1} a_{t+2} \cdots a_{t'-1} a_{t'}=\emptyset$
Job Training
Job Training
0 1 2⋯ t-1 t
t+1 t+2 ⋯t'-1 t'
D_0
D_t
$D_{t'}$
Condition (1) or (2) is reached
A new schedule is required
Condition (1) or (2) is reached

Figure 3: Time steps in the system.

Policy. A policy $\pi(a|s)$ indicates the probability distribution over all actions $a \in \mathcal{A}$ given a state s. Let S' be the subset of S that contains the states satisfying Condition (1) or Condition (2), and the state that indicates there is no workload currently in the cluster. At any state $s \in S'$, a void action $\emptyset$ should be taken, and therefore the probability distribution for the action over the set $\mathcal{A}$ is $(0, \cdots, 0, 1)$. As for the rest of the states, the action distribution is given by $\pi'(a|s;\theta)$, which is approximated by a DNN with learned parameters θ to be described in Section 4.3. In conclusion, the policy of *Chic* could be represented as

$$\pi(a|s;\theta) = \begin{cases}\pi'(a|s;\theta), & s \in S - S';\\ (0,\cdots,0,1), & s \in S'.\end{cases} \tag{6}$$

State Transition. After taking an action $a \in \mathcal{A}$, the state of the system is changed. The time steps in the system are illustrated in Fig. 3. During one decision making phase, assume at time step t the DRL agent takes action $a_t = 2i-1, i \in [I]$, which means increasing the allocation of worker nodes to workload i by 1. Then $s^w_{i(t+1)}$ will transit to $s^w_{it} + 1$ if $s^B_{it} > 0$. Otherwise, $s^w_{i(t+1)}$ remains 0 if no ML workload arrives during time step t. A similar transition is applied to s^p_{it} when $a_t = 2i, i \in [I]$. The duration D_t of each time step during the decision making phase depends on the processing time of the DRL agent, i.e., the time used by the DRL agent to give an action a_t according to the input state s_t and update the state to s_{t+1}. In our experiment, for a DRL agent with a DNN policy approximator composed of 3 hidden layers build in the PyTorch framework running on a single Intel i5 3.1 GHz CPU core, the processing time of each time step requires no more than 0.78 ms.

The time steps proceed until either Condition (1) or Condition (2) is reached, and then the system enters the job training phase. The entire job training phase is counted as one time step in our system. The duration D_t of the time step corresponding to the job training phase is influenced by the earliest finish time of the I workloads and the arrival time of new ML workloads. When there are no fewer than I workloads in the cluster, D_t is determined by the earliest finish time of the I workloads, which could be represented as $\min_i \frac{s^B_{it}}{g_i(s^w_{it}, s^p_{it})}$. Otherwise, D_t is decided by the shorter duration between the earliest completion of I_t workloads and the arrival of a new ML workload.

After the job training phase, $s^w_{i(t+1)}$ and $s^p_{i(t+1)}$ are reset to 0 for each $i \in [I]$ in order to determine a new schedule in the subsequent time steps. At this time step, the number of remaining iterations for the first I workloads could be expressed as

$$s^B_{i(t+1)} = \max\{0, s^B_{it} - \left\lfloor \frac{D_t}{g_i(s^w_{it}, s^p_{it})} \right\rfloor\}. \tag{7}$$

If $s^B_{i(t+1)}$ equals zero, the i-th ML workload is finished and will be removed. Meanwhile, the first workload waiting in the queue will be revealed to the DRL agent at time step $t + 1$.

Reward. The objective of *Chic* is to obtain a scheduling decision to minimize the average completion time of ML workloads. Let $R = \sum_{t\in[T]} r_t$ denote the cumulative reward from the first time step, where r_t is the reward received after taking action a_t at state s_t. We note that r_t should be designed and chosen carefully so that maximizing the cumulative reward R mimics achieving the objective of *Chic*. We design the reward r_t as the sum of three parts: r_{t1}, r_{t2}, and r_{t3}. The average completion time is reflected by r_{t1}:

$$r_{t1} = -(\sum_{i\in[I]} D_t \mathbf{1}_{\{s^B_{it}>0\}} + N_t D_t + \sum_{j\in[J_t]} (D_t - A_j)), \tag{8}$$

where $\mathbf{1}_{\{s^B_{it}>0\}}$ is the indicator function and J_t denotes the number of workloads arriving during time step t. Each workload $j \in [J_t]$ has an arrival time A_j counting from the beginning of time step t. Therefore, by summing the three terms and negating the value, r_{t1} is the negative of the sum of the time that all workloads spend in the system in time step t.

The introduction of r_{t2} and r_{t3} is to reduce the likelihood of unreasonable actions. Firstly, because the DRL agent should not allocate resources to a nonexistent workload, a punishment $r_{t2} = -p_1$ is added when $a_t = 2i - 1$ or $2i$, and $s^B_{it} = 0$. The punishment value $-p_1$ should be large enough to guide the DRL agent in avoiding this action at such states. Otherwise, r_{t2} is set as 0. Similarly, a situation where a workload has no fewer than 1 worker (resp. PS) nodes but 0 PS (resp. worker) node is not favorable since it wastes resources with no progress for that workload. Hence, a punishment $r_{t3} = -p_2$ is added when the DRL agent tries to allocate resources for workload i for the first time. After allocating both workers and PSs for workload i, the punishment is removed by adding $r_{t3} = p_2$. With r_{t2} and r_{t3}, the cumulative reward R does not only represent the negative of the total time that all workloads spend in the system. But by continuing to learn, the DRL agent should decrease the occurrence of unreasonable actions, and the reward will therefore mainly reflect the average completion time.

4.3 Training Algorithm

The starting point of the training of *Chic* is the cross-entropy method. However, we have found that directly applying the cross-entropy method does not lead to satisfactory results. The reason is that the initialization of the parameters $\tilde{\theta}_0$ in the DNN approximating $\pi'(a|s;\theta)$ will influence the finding of the optimal importance sampling density. For a policy $\pi(a|s;\tilde{\theta}_0)$ with high probability densities on actions leading to trajectories with a low cumulative reward, the sampling efficiency will be extremely low. In that case, obtaining a sample x_n with high $R(x_n)$ typically requires a large number of samples. However, increasing the number of samples means extending the training time. Furthermore, some actions might never be sampled at a specific state due to the initialization $\pi'(a|s;\tilde{\theta}_0)$. Since some state-action pairs never occur, the policy might be stuck in a local optimum. Therefore, at iteration k, except N samples from the current importance sampling density, a sample x_{N+1} obtained by replacing $\pi'(a|s;\tilde{\theta}_{k-1})$ with a uniform distribution on action space $\mathcal{A}$ is introduced. In this case, the estimate is:

$$\begin{aligned}\hat{\theta}_k = \arg\max_{\theta_k}[&\sum_{n\in[N]} \mathbf{1}_{\{R(x_n)\geq\hat{\xi}_k\}} \sum_{a_t,s_t\in x_n} \pi'(a_t|s_t;\theta_k) \\ &+ \mathbf{1}_{\{R(x_{N+1})\geq\hat{\xi}_k\}} \sum_{a_t,s_t\in x_{N+1}} \pi'(a_t|s_t;\theta_k)].\end{aligned} \tag{9}$$

When $\pi'(a|s;\tilde{\theta}_{k-1})$ has a high probability for choosing the right action a at state s, the sampled trajectories $x_n, n \in \{1,\cdots,N\}$ are more likely to have higher cumulative rewards and $\mathbf{1}_{\{R(x_{N+1})\geq\hat{\xi}_k\}}$ is less likely to be 1. In this case, the second term has a higher probability to be 0 and the updating rule is the same as not introducing x_{N+1}. On the contrary, when the importance sampling density is far from optimal, sampling trajectories by following policy $\pi(a|s;\tilde{\theta}_{k-1})$ might not provide better trajectories than choosing the action randomly at each state $s \in S - S'$. Therefore, $s_t, a_t \in x_{N+1}$ will have a higher probability to be leveraged to update θ. Meanwhile, the sample x_{N+1} obtained by choosing actions uniformly distributed on action space $\mathcal{A}$ will decrease the likelihood that some actions are never executed at a certain state $s \in S - S'$.

Algorithm 1 *Chic* Training Algorithm

```
1:  Randomly initialize parameter $\tilde{\theta}_0$ in $\pi(a|s;\tilde{\theta}_0)$
2:  for iteration $k = 1$ to $K$ do
3:      for episode $n = 1$ to $N$ do
4:          Randomly select a workload arrival sequence
5:          Generate $x_n$ with current policy $\pi(a|s;\tilde{\theta}_{k-1})$
6:      end for
7:      Generate $x_{N+1}$ by selecting actions with uniform distribution on $\mathcal{A}$ for each state $s \in S - S'$
8:      Calculate the cumulative reward $R(x_n)$ for all $n$, and order them from smallest to largest, $R_{(1)} \leq \cdots \leq R_{(N+1)}$
9:      Let $\hat{\xi}_k = R_{(\lfloor(1-\rho)N\rfloor)}$
10:     Calculate $\hat{\theta}_k$ by solving (9)
11:     Update $\tilde{\theta}_k$ as $\tilde{\theta}_k = \alpha\hat{\theta}_k + (1-\alpha)\tilde{\theta}_{k-1}$
12: end for
```

To train a policy that generalizes, we consider multiple examples of workload arrival sequences. At the beginning of each episode, a sequence is chosen randomly from the examples. In each iteration, $N+1$ episodes are taken. We formally present the training algorithm of the DRL-based scheduling framework *Chic* as Algorithm 1. The algorithm is mainly composed of two iterative phases. At each training iteration k, it generates N random samples of trajectories by following the current policy $\pi(a|s;\tilde{\theta}_{k-1})$, and one trajectory by choosing actions according to a uniform distribution on action space $\mathcal{A}$ at each state $s \in S - S'$, which is shown in Lines 3-7. Then, it updates the parameters of the policy based on the $\lfloor(1-\rho)N\rfloor$ best performing trajectories using cross-entropy minimization and a smooth update with parameter α, which is shown in Lines 8-11.

4.4 The Chic-Pair Simplification

A common characteristic of most DRL jobs is that the training of a policy typically requires a substantial amount of time. A smaller action space usually leads to a shorter training time since the range the DRL agent has to explore is narrowed. In this section, we investigate whether the action space of *Chic* could be further simplified.

From the schedule given by *Chic* in our experiments, we found the number of worker nodes and PS nodes assigned to a workload is similar. Besides, setting the amount of PS nodes the same as the number of worker nodes is not unusual in distributed learning, and is in fact also one default case in MXNet [9]. Therefore, we propose a simplified version of *Chic*, named *Chic-Pair*, with a simpler action space.

Chic-Pair allocates worker nodes and PS nodes in a pair. More specifically, the action a_t in *Chic-Pair* is designed as the ML workload to which a worker node and a PS node is added, so that $a_t = i$ refers to allocating both a worker node and a PS node to ML workload i at time step t, where $i \in [I]$. The action space for a system scheduling for the first I workloads is therefore given by $\mathcal{A} = \{1, \cdots, I, \emptyset\}$. *Chic-Pair* reduces the size of the action space from $2I + 1$ to $I + 1$ compared with *Chic*. Furthermore, since s_{it}^{w} always equals s_{it}^{p}, the state could be simplified by removing s_{it}^{p}. Condition (1) should be changed correspondingly to there existing a type $k \in [K]$ resource such that $\min_i(w_i^k + p_i^k) + \sum_{i \in [I]} s_{it}^{w}(w_i^k + p_i^k) > c_k$. Moreover, since *Chic-Pair* allocates a pair of worker node and PS node at each time step, the concern that a workload is allocated with at least one worker (resp. PS) but zero PS (resp. worker) is no longer present, and r_{t2} could be eliminated. The other design components of *Chic-Pair* remain the same as those of *Chic*.

5 IMPLEMENTATION

We use PyTorch to implement *Chic* and *Chic-Pair* running the modified cross-entropy method with a policy network. There are a few hyper-parameters in the proposed scheduling framework. To maximize its performance, comprehensive empirical training sessions are conducted to find the best settings for them and the best structure for the policy network. A fully connected feed-forward neural network with 3 hidden layers serves as $\pi'(a|s; \theta)$. The neural network includes 64, 32, and 4 neurons in the first, second, and third hidden layers respectively and utilizes the ReLu function for activation. A dropout rate of 0.5 is introduced in the first hidden layer. The output of the neural network is passed to the softmax function to ensure the sum of output values equals to 1. We have found that more complicated network structures with more hidden layers and more neurons in each layer take longer time to train and do not perform much better than the chosen structure.

During the training phase of the DRL agent, $N = 35$ trajectories are sampled with the current policy in each iteration k and 0.3-quantile of the performance of the sampled trajectories is utilized as $\hat{\xi}_k$. The Adam optimizer is used with learning rate $\alpha = 0.01$. Furthermore, a validation set containing 30 workload sequences is leveraged to find the best learned policy. *Chic* and *Chic-Pair* go through the validation set every two iterations and record the average completion time achieved by the current policy. At the end of the training phase, the policy with the shortest average completion time for the validation set is chosen for testing.

6 PERFORMANCE EVALUATION

We conduct extensive simulations of *Chic* and *Chic-Pair* to show their effectiveness in scheduling workloads in ML clusters.

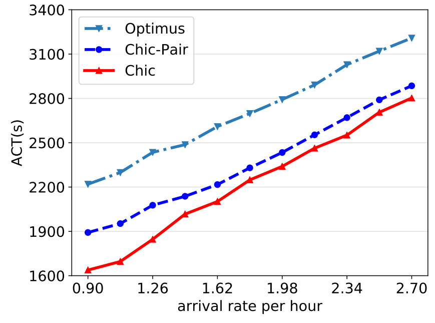
6.1 Simulation Settings

Baseline: We compare *Chic* and *Chic-Pair* with *Optimus*, which is the state-of-the-art approach in the literature for scheduling ML workloads in the cluster. *Optimus* schedules ML workloads with fixed time intervals and keeps assigning one worker node or one PS node to an ML workload based on some marginal gain until at least one type of resource is used up, or all the marginal gains are negative. It has been shown that *Optimus* outperforms DRF and Tetris, which are general cluster schedulers but not explicitly designed for ML workloads [4].
Workload: The arrival pattern for ML workloads follows a Poisson arrival process. The arrival rate is chosen between $0.9\ \text{h}^{-1} \sim 2.7\ \text{h}^{-1}$. Each ML workload is randomly chosen from 3 types of models: ResNet-50 or VGG-16 on a downscaled ImageNet ILSVRC2012 dataset, or ResNext-110 on a part of the Cifar-10 dataset. The VGG-16 model requires 1 GPU, 2 CPU cores, and 10GB memory for each worker and 4 CPU cores and 10GB memory for each PS. A worker of the ResNet-50 model requires 1 GPU, 2 CPU cores, and 8GB memory while a PS requires 3 CPU cores and 9GB memory. For the Resnext-110 model, a worker needs 1 GPU, 2 CPU cores, and 10GB memory, while a PS needs 3 CPU cores and 10GB memory. Each workload is trained for a number of iterations that is uniformly distributed between 100 and 200, and the batch size for the gradient update discussed in Section 2.1 is 32.
Settings: We simulate a cluster with 10 GPUs, 120 CPU cores, and 600GB memory. During the training of *Chic* and *Chic-Pair*, I is set to be 3 by default, but we also study other choices of I. During the training phase, the DRL agent schedules $W = 30$ workloads in each episode. The default schedule interval of *Optimus* is set as 600 secs, which is indicated in [4]. The reward r_{t1} in (8) is calculated with a unit of 20 mins. We observe that only the relative difference among r_{t1}, r_{t2}, and r_{t3} matter. Therefore, we have fixed r_{t1} and experimented with different values of of p_1 and p_2 to inspect the working conditions of *Chic*. Our results indicate that $p_1 = 1$ is a large enough punishment. Furthermore, a wide range of p_2 values is observed to work well, further demonstrating the stability of *Chic*. In the following, we present simulation results with p_1 and p_2 set to 5 and 1 respectively.

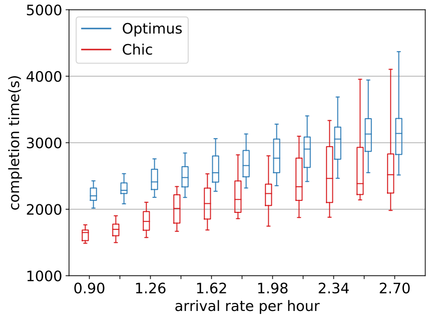
Our goal is to find the performance of *Chic* compared with a state-of-the-art scheduler and the applicability of *Chic* under different conditions. Furthermore, we study whether the simplified version of *Chic*, i.e., *Chic-Pair*, could produce satisfactory results.

6.2 Simulation Results

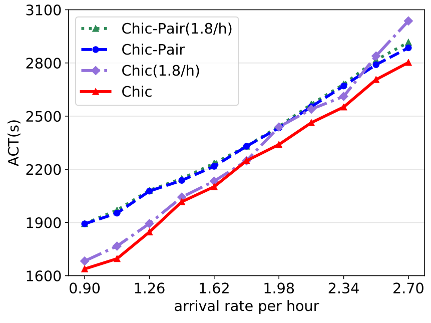
6.2.1 Average Completion Time (ACT). First, we compare the ACT of ML workloads achieved by *Chic* and *Chic-Pair* with *Optimus*. The results are shown in Fig. 4. Each data point in Fig. 4(a) and 4(c) is an average of 30 examples of workload sequences not used during training. From Fig. 4(a) we can see that *Chic* performs the best, reducing the ACT of *Optimus* by more than 400 secs in all cases. *Chic* and *Chic-Pair* decreases the ACT by at least 12.65% and 10.08% respectively compared with *Optimus*. As expected, *Chic* achieves



(a) *Chic*, *Chic-Pair*, and *Optimus*.

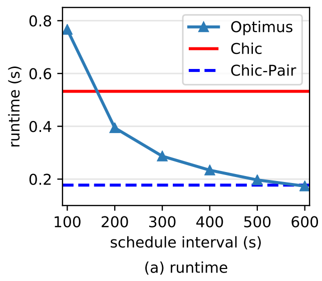


(b) *Chic* and *Optimus*.



(c) Training with mismatched arrival rate.

Figure 4: Average completion time.



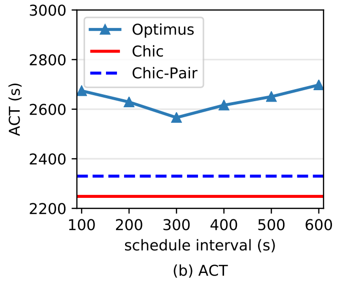


Figure 5: The runtime and ACT of *Chic*, *Chic-Pair*, and *Optimus*.

better performance than *Chic-Pair* since it allows the DRL agent to decide the allocation of both worker nodes and PS nodes. Fig. 4(b) provides a more detailed performance comparison between *Chic* and *Optimus*. *Chic* outperforms *Optimus* in terms of the shortest completion time and the median completion time. In addition, *Chic* is comparable to and often better than *Optimus* in terms of the longest completion time.

Fig. 4(c) reveals the applicability of *Chic* and *Chic-Pair*. The policy of *Chic* is re-trained under different workloads while *Chic*(1.8/h) and *Chic-Pair*(1.8/h) are only trained once under an arrival rate of 1.8 h^{-1}. An interesting observation is that *Chic*(1.8/h) almost performs as well as *Chic* when the arrival rate is not high, and the difference in ACT is no more than 4.06%. Though the difference in ACT obtained by *Chic*(1.8/h) and *Chic* is larger when the arrival rate is high, it shows favorable properties of the experience-driven approach. When there is no dramatic change in the workloads within the cluster, it is not necessary to take extra time to re-train the DRL agent of *Chic*. Furthermore, *Chic-Pair*(1.8/h) shows even better adaptivity than *Chic*(1.8/h). The reason is that the action space of *Chic-Pair* is simplified and the DRL agent only needs to determine one dimension of the schedule. Therefore, the schedule obtained by *Chic-Pair* has less variance. Combining Fig. 4(a) and 4(c) we can see that the learned policy of *Chic* is more accurate and specific while the policy of *Chic-Pair* is more general.

We further examine the runtime of *Chic* and *Chic-Pair*. We focus on the data point with an arrival rate of 1.8 h^{-1}. The measurement is conducted on a Macbook Pro laptop with 3.1 GHz Intel Core i5. As can be seen from Fig. 5(a), the runtime of *Chic* to schedule 30 workloads is 3.00 times longer than *Chic-Pair*. Meanwhile, the run time of *Chic-Pair* is similar to *Optimus* when the schedule interval is set as 600 secs, the same as in [4]. However, when we decrease the schedule interval of *Optimus*, *Chic-Pair* outperforms *Optimus* in runtime. Though the best ACT achieved by *Optimus* is when the schedule interval is decreased to 300 secs, as shown in Fig. 5(b), the runtime increases while the ACT still cannot catch up with *Chic* and *Chic-Pair*. The time *Chic* used to determine the schedule is short compared with the decrease in ACT. In general, *Chic* is a good choice for an ML cluster aiming for the shortest ACT while *Chic-Pair* is ideal when a fast schedule is required. Furthermore, multiprocessing or offloading to GPU servers could further decrease the runtime of *Chic* and *Chic-Pair*.

6.2.2 Convergence Behavior. To understand the convergence behavior of *Chic* and *Chic-Pair*, we study in detail the training of *Chic* and *Chic-Pair* to optimize the ACT at an arrival rate of 1.8 h^{-1}. Fig. 6(a) plots the bound $\hat{\xi}_k$ and the average of $R(x_n)$ among all the sampled 36 trajectories by the end of each iteration k during the training of *Chic*. As expected, both values increase with iterations as *Chic*'s policy improves. Fig. 6(b) illustrates the ACT achieved by the learned policy of *Chic* at the end of each iteration k. To compare, the figure also shows the ACT achieved by *Optimus*. From Fig. 6(b) we see that *Chic* improves with iteration counts. In particular, after 19 iterations *Chic* is better than *Optimus*. The improvement in the first several iterations is huge but gradually the change slows down. In addition, the gap between the reward bound $\hat{\xi}_k$ and the average of $R(x_n)$ among all the sampled trajectories narrows down, meaning the model is steadily converging. Furthermore, with both Fig. 6(a) and 6(b), we can see that higher cumulative reward, which is what the training algorithm optimizes for, correlates to lower ACT, which is the objective of *Chic*. This correlation demonstrates that the design of our reward to achieve the optimization objective for the scheduling of ML workloads is reasonable.

The training of each iteration in *Chic* is around 28.15 secs on one core of 3.1 GHz Intel i5 CPU. The policy is only required to be

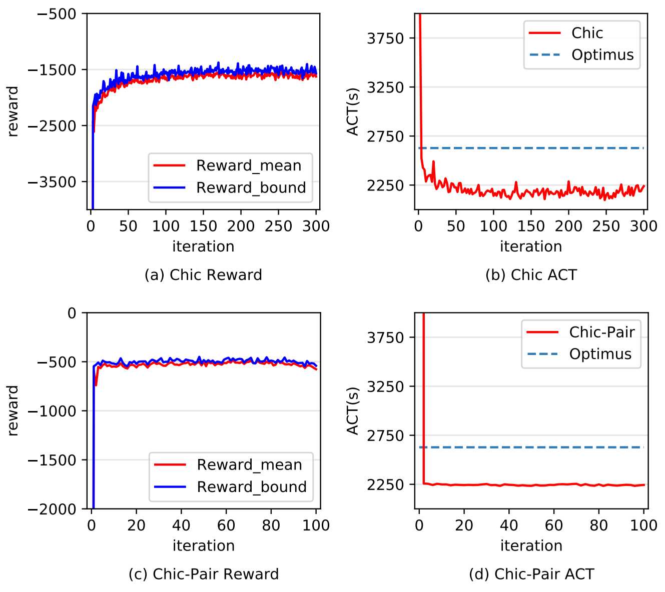


Figure 6: The Learning Curve of *Chic* and *Chic-Pair*.

trained once to schedule ML workloads. Retraining is only needed when the workloads change greatly as discussed in Section 6.2.1. Similar training results are obtained for *Chic-Pair* as shown in Fig. 6(c) and 6(d). Moreover, *Chic-Pair* converges more quickly than *Chic* due to the simplicity of the policy.

6.2.3 The influence of W and I. We further discuss the influence of the selection of W and I. Larger W leads to a longer duration of each episode but also increases the probability that the cluster is in steady state during the training. Fig. 7(a) plots the influence of W on *Chic* and *Chic-Pair*. Adopting $W = 60$ yields a slightly better result, but the training time is much longer. Therefore, $W = 30$ is a suitable choice in general.

The selection of I should be combined with the arrival rate. When the arrival rate is high but I is small, many workloads will be waiting in the queue and the DRL agent will only focus on the scheduling of the first several arrived workloads. In contrast, when the arrival rate is low but I is large, the larger action space requires the DRL agent to take more time to explore. Because the training of *Chic* takes too much time, the simulation is conducted for *Chic-Pair*. Fig. 7(b) plots the ACT when the size of I is $2, 3$, or 4. As can be seen from the figure, when the arrival rate is 0.9 h^{-1}, which is not high, the ACT achieved for different I values is similar. However, when the arrival rate is 2.7 h^{-1}, setting $I = 2$ limits the performance of *Chic-Pair* since more workloads are queuing to be scheduled. Compared with the case when the DRL agent schedules for the first 3 workloads, setting $I = 4$ could decrease the ACT by 71.28 secs.

7 RELATED WORK

A growing amount of research attention has been given to distributed ML due to the increasing complexity and data sizes of ML workloads. In a distributed ML cluster, various training workloads with different models and huge datasets join in an online

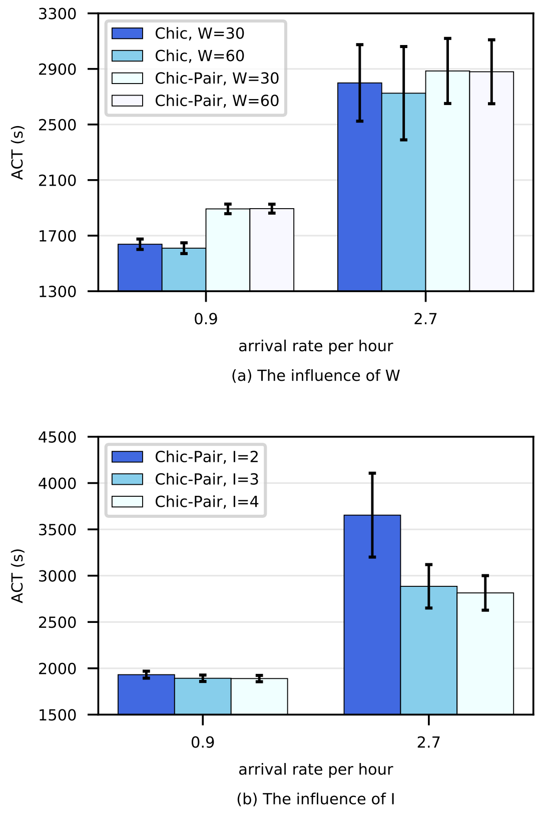


Figure 7: The Influence of W and I.

fashion. As ML workloads are typically resource-intensive and time-consuming even with distributed learning, it is important to manage the resources in an ML cluster efficiently to satisfy training requirements.

There has been a significant amount of effort on the study of general cluster resource allocation to achieve different performance objectives. Mesos [14] and YARN [3] use the DRF [15] strategy to allocate resources. Borg [2], Fuxi [16], and Firmament [17] support policy-based scheduling. However, all of these schedulers are designed for general purposes.

To further leverage the physical resources and expedite the training process, some researchers considered leveraging the characteristics of ML workloads and designing scheduling algorithms that are tailored for ML workloads. Bao *et al.* designed OASiS [10], which assigns a changing number of worker nodes and PS nodes to an ML workload upon its arrival based on a carefully designed price function. However, the exact value of the price could not be obtained without full knowledge of all incoming workloads. Furthermore, OASiS is designed with the assumption that worker nodes and PS nodes are deployed on two different sets of physical servers, limiting the use cases.

Zhang *et al.* designed SLAQ [18], a scheduling system for approximating ML training workloads to maximize system-wide quality improvement. The authors targeted the training quality of experimental ML models instead of models in production. Besides, SLAQ only works for the single-resource case. *Optimus* [4] made several further improvements in this direction. *Optimus* adjusts the deployment of both worker nodes and PS nodes for ML workloads with fixed time intervals. The key idea is to iteratively add a worker node or a PS node to the workload with the most decrease in completion time with such resources. The predicted completion time is based on the assumption that the schedule for the workload remains the same. However, since *Optimus* has to change the schedule from time to time to suit system dynamics, it might not be effective in the long run. Also, it assigns at least 1 worker node and 1 PS node to each of the workloads, causing inefficiency when there are many workloads in the cluster. Though possessing some drawbacks, all of these methods showed better performance compared with general schedulers, proving the effectiveness of schedulers that are designed specifically for ML workloads.

The success of applying DRL to play video games [5] and ComputerGo [19] inspired some researchers to leverage DRL to solve resource management problems. DeepRM [6] is an attempt in this direction by using the policy gradient method to determine the time for a job to be scheduled. Different from our problem, DeepRM only determines the time slots for jobs to be scheduled, and the resource requirements and occupancy time of a job are all known conditions. Mirhoseini *et al.* employed DRL to expedite the training of a single machine learning workload [7]. Rather than data parallelism, they dealt with the model parallelism case and let the DRL agent decide the mapping between each part of the model and devices. These attempts have shown the potential of utilizing DRL techniques to solve challenging resource management problems.

8 CONCLUSION

In this paper, we have studied the scheduling of ML workloads in cloud computing clusters. Different from previous schedulers using heuristics to address this problem, we wondered whether an ML cluster can learn to manage the resources by itself via leveraging its experience. Inspired by the success of applying DRL in recent years, we have developed a new scheduler *Chic* to assign worker nodes and PS nodes to ML workloads with the objective of minimizing the average completion time. With an appropriate design of the state, action, and reward, *Chic* enables experience-driven scheduling by learning the environment dynamics. Moreover, a simplified version named *Chic-Pair* is proposed to shorten the training time. Our extensive array of simulation results have shown clear evidence that *Chic* and *Chic-Pair* are able to significantly outperform a state-of-the-art heuristic, *Optimus*, in scheduling ML workloads under a wide variety of conditions.

9 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments on improving this paper. This work has been funded in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

REFERENCES

[1] M. Li, D. G. Anderson, J. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[2] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (Eurosys)*, 2015.

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

[4] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the European Conference on Computer Systems (Eurosys)*, 2018.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, February 2015.

[6] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2016.

[7] A. Mirhoseini, H. Pham, Q. L., M. Norouzi, S. Bengio, B. Steiner, Y. Zhou, N. Kumar, R. Larsen, and J. Dean, "Device placement optimization with reinforcement learning," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2017.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[9] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," in *Proceedings of the International Conference on Neural Information Processing Systems (NIPS) Workshop on Systems for Machine Learning and Open Source Software (LearningSys)*, 2015.

[10] Y. Bao, Y. Peng, C. Wu, and Z. Li, "Online job scheduling in distributed machine learning clusters," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2018.

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

[12] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, October 2017.

[13] R. Rubinstein and D. Kroese, *The Cross-Entropy Method.* Springer, 2004.

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[15] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[16] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: A fault-tolerant resource management and job scheduling system at internet scale," in *Proceedings of the VLDB Endowment (PVLDB)*, 2014.

[17] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand, "Firmament: Fast, centralized cluster scheduling at scale," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[18] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "SLAQ: Quality-driven scheduling for distributed machine learning," in *Proceedings of the Symposium on Cloud Computing (SoCC)*, 2017.

[19] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, January 2016.