Cloud Resource Scheduling With Deep Reinforcement Learning and Imitation Learning

Wenxia Guo, Wenhong Tian, Yufei Ye, Lingxiao Xu, and Kui Wu, *Senior Member, IEEE*

***Abstract*—The cloud resource management belongs to the category of combinatorial optimization problems, most of which have been proven to be NP-hard. In recent years, reinforcement learning (RL), as a special paradigm of machine learning, has been used to tackle these NP-hard problems. In this article, we present a deep RL-based solution, called DeepRM_Plus, to efficiently solve different cloud resource management problems. We use a convolutional neural network to capture the resource management model and utilize imitation learning in the reinforcement process to reduce the training time of the optimal policy. Compared with the state-of-the-art algorithm DeepRM, DeepRM_Plus is 37.5% faster in terms of the convergence rate. Moreover, DeepRM_Plus reduces the average weighted turnaround time and the average cycling time by 51.85% and 11.51%, respectively.**

***Index Terms*—Cloud resource scheduling, deep reinforcement learning (deep RL), imitation learning.**

I. Introduction

CLOUD computing has become the foundation for many business applications [1], [2]. Based on virtualization technologies, the cloud service provider can dynamically allocate physical resources to meet various service-level agreements (SLAs) of end users. To maximize the profit, the cloud service provider needs to support users' service requests as many as possible over the resource and SLA constraints. As such, efficient resource management is critical to the success of any cloud service provider.

Resource management is not new and has been broadly studied in many systems, including, for example, memory management [3] and job scheduling in computers [4], and congestion control in networks [5]. Resource management in cloud computing, however, is extremely challenging due to several reasons. First, the resource demands are diverse and dynamic. Service requests[1] may demand multiple types of resources, and the "holding time" of each resource may be different. The service interference on the shared resources is hence much more complex. Second, due to the special business model of cloud computing (e.g., the pay-as-you-use model), decisions for resource allocation must be made online with uncertain future request arrivals. Finally, with the fast development of big data and cloud computing, the scale of resources, operations, and processes in the cloud continues to expand. Often than not, resource scheduling in a data center needs to schedule resources for tens of millions of jobs.

Optimal resource allocation generally falls in the category of combinatorial optimization problems, many of which have been proven to be NP-hard or NP-complete [6], [7]. Traditionally, the way to tackle these problems uses approximation or heuristics. Based on the empirical studies, many heuristics have been proposed, for example, *first-fit*, *best-fit*, shortest-job-first (SJB), and so on. Clearly, the performance of heuristics depends on not only the statistical patterns of the resource demands but also the constraints on multiple resources. If the underlying scenarios change, a well-performed heuristic may behave poorly. *In principle*, the resource scheduler should dynamically reconstruct new heuristics to align with the underlying system changes. Nevertheless, due to the large scale and the complexity of resource management in the cloud, it is extremely hard to achieve this ideological principle in practice.

The fast advance in machine learning, particularly in reinforcement learning (RL), offers new opportunities to tackle the above challenges [8]. In a nutshell, inside the core of RL, an agent (i.e., the decision maker) interacts with the environment and learns automatically from the experience based on the reward from the environment. RL uses the framework of the Markov decision process (MDP), where the agent chooses an action at each state and receives a reward indicating the quality of the executed action. The ultimate goal of the agent is to learn a mapping (also called a policy) between the state of the environment and the behavior so that the agent can select a series of optimal actions to obtain the maximum cumulative reward from the environment. Such a mapping is built gradually based on trial and error. This idea of *learning from experience* makes RL a promising method for cloud resource scheduling [9], [10], because we normally do not have a predefined model to capture the dynamics and complexity in cloud resource management.

Manuscript received March 23, 2020; revised June 29, 2020 and August 4, 2020; accepted September 2, 2020. Date of publication September 18, 2020; date of current version February 19, 2021. This work was supported in part by the National Natural Science Foundation of China under Project 61672136 and Project 61828202, and in part by the National Key Research and Development Plan under Award 2018AAA0103203. *(Corresponding authors: Wenhong Tian; Kui Wu.)*

Wenxia Guo, Wenhong Tian, and Lingxiao Xu are with the School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu 610054, China (e-mail: tian_wenhong@uestc.edu.cn).

Yufei Ye was with the School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu 610054, China.

Kui Wu is with the Department of Computer Science, University of Victoria, Victoria, BC V8P 5C2, Canada (e-mail: wkui@uvic.ca).

Digital Object Identifier 10.1109/JIOT.2020.3025015

[1]We use service requests, jobs, and tasks interchangeably in this article.

Traditionally, *learning from experience* can be implemented in a tabular manner: the agent keeps a table (e.g., Q table [11]) to track and record the expected return at each state. This type of RL works great for small-scale problems [12] but may not be practical when the number of states is huge or the states are continuous. One way to overcome this problem is to use an approximate function rather than a table to capture the relationship between input states and output actions. Such an approximate function could be built with deep neural networks, and in this case, we call the method deep reinforcement learning (deep RL) [13]. Deep RL has been used successfully not only in computer games, such as the famous Atari [14] and AlphaGo [15], [16] but also in resource management [17].

DeepRM [17] is so far the most representative deep RL-based solution for resource management. When applying DeepRM for cloud resource scheduling, however, we have found that: 1) it suffers from very long training time and slow convergence rate and 2) its convolutional neural network (CNN) is too simple to capture the complex state–action relationship in data centers. This is mainly because DeepRM is completely ignorant of "expert" knowledge and let RL learn the optimal policy from scratch with trial and error. This observation motivates us to develop a more efficient deep RL solution for cloud resource management, called DeepRM_Plus. Our contributions are summarized as follows.

1) To speed up model training and convergence rate, we integrate imitation learning in deep RL to imitate the behavior of some classical heuristics (e.g., SJB), which have good properties with respect to a given metric (e.g., the least average waiting time). Those heuristics are considered as expert for specific system scenarios (i.e., demand patterns and available resources). Since we normally do not know in advance the exact system state, the goal of imitation learning is not to copy the exact heuristic strategy, but instead to instill the expert knowledge in the RL process. By first trying expert strategies rather than taking some blind actions, we can significantly speed up the learning speed.
2) Besides imitation learning, DeepRM_Plus is also significantly different from DeepRM in the architecture of the deep network and search space. The new CNN in DeepRM_Plus makes it better capture the state–action characteristics of the data center.
3) We perform an extensive evaluation of DeepRM_Plus and compare its performance with that of several baseline resource scheduling schemes, including *Random*, first-come–first-serve (FCFS), shortest job first (SJF), highest-response-ratio-next (HRRN), Tetris [4], as well as the state-of-the-art DeepRM. Experimental results demonstrate that DeepRM_Plus outperforms all the above resource scheduling schemes.

The remainder of this article is organized as follows. Related work is discussed in Section II. The background information on RL is introduced in Section III. In this section, we also describe the system assumptions and formally define two cloud resource scheduling problems. In Section IV, we present the details of DeepRM_Plus. The experimental environment and the evaluation results are given in Section V. Finally, we summarize and conclude this article in Section VI.

II. Related Work

Work related to this article can be roughly grouped into two categories: 1) cloud resource management and 2) machine learning-based resource management.

In the first category, the problem has been investigated from various angles that can be roughly classified as: 1) virtual machine (VM) allocation and consolidation [18]–[20]; 2) integrated resource scheduling and load balancing [21]–[23]; and 3) task assignment to maximize resource utilization [24], [25]. Since the first two groups of research are not comparable to this work, we only focus on introducing the third group as follows.

Regarding the work in task assignment for maximum cloud resource utilization, Tetris [4] was proposed to handle multiresource packing in the cluster scheduler. It prefers jobs with shorter job duration so that the spare cluster can execute as many jobs as possible. Nevertheless, this method has a poor performance when the cluster load is heavy, since, in this case, the short jobs have to wait for a long time before the long jobs finish.

Based on the concept of the intuitionistic fuzzy set theory, Raheja [26] proposed an intuitionistic fuzzy-based HRRN scheduler. This approach can deal with the impreciseness of the response ratio and adapt the schedules based on continuous feedback. A dynamic task assignment scheduling scheme, called EFDTS [27], was developed to optimize resource utilization with a fault-tolerant mechanism.

In the second category, machine learning methods have been broadly used in resource management. Due to similar reasons as above, we only focus on the task assignment problem for maximum resource utilization. Nevertheless, depending on different machine learning techniques used in the solution, research in this domain can be classified in different ways, including: 1) bioinspired solutions [28]; 2) Q-learning-based solutions [29]–[31]; and 3) deep RL-based solutions [17], [32].

Regarding bioinspired solutions, Domanal *et al.* [28] proposed three bioinspired algorithms, modified particle swarm optimization (MPSO), modified cat swarm optimization (MCSO), and HYBRID (MPSO+MCSO), to make better assignments of tasks. In addition, HYBRID overcomes the limitations of the first two methods and decreases the total execution time and communication overhead between VMs and resource pool.

In terms of Q-learning-based solutions, Farahnakian *et al.* [29] proposed a dynamic consolidation method based on Q-learning to allocate incoming requests. The agent learns from the previous knowledge to make the best decision on whether a host should be active or fall asleep. Dab *et al.* [33] proposed a new joint task assignment and resource allocation approach, named QL-joint task assignment and resource allocation (QL-JTAR), to minimize the latency.

As for deep RL-based solutions, Mao *et al.* [17] proposed a system, named DeepRM, which efficiently manages the

nonpreemptive scheduling of online jobs with deep RL. In this method, the information on resource utilization and the states of jobs to be scheduled are handled by a fully connected neural network. Then, according to the output of the neural network, RL is adopted to manage jobs in the cluster. However, DeepRM needs a long training process and the architecture of the neural network is not conducive to extracting the complex state–action relationship in data centers.

III. Background

To help better understand the article and to make the article self-contained, we briefly review some background techniques. More detailed introduction can be found in [11], [34], and [35].

A. Reinforcement Learning

RL is an unsupervised learning method. The RL framework generally consists of an agent and the environment. The agent interacts with the environment and takes "educated" actions based on the reward received from the environment. The system is modeled as an MDP. The agent observes the state changes and based on the current system state and the received rewards, it makes decisions to achieve some objective, e.g., maximizing the expected total return. The agent learns the optimal policy by trial and error from the interactions with the environment. Formally, the main components of RL include the following.

1) *State Space S:* A set of system states.
2) *Action Space A:* A set of actions that the agent can perform.
3) *Environment:* Everything outside of the agent that interacts with the agent. The environment receives a series of actions from the agent. Through evaluating the quality of these actions, the environment gives back the corresponding reward of each action.
4) *Reward:* The feedback signal of the environment to the action. In this article, we consider the signal as a value of the objective function, and the agent aims to minimize the function value during the learning process.
5) *Policy π:* A mapping between the environment state and the action. At each time step, the agent determines the next action based on the observation in the current state.

RL algorithms can be divided into value-based RL (e.g., Q-learning) and policy-based RL (e.g., policy-gradient). Simply put, value-based RL estimates the expected return (the so-called Q value) of each state and takes optimal decisions based on the Q values. As we have discussed in Section I, value-based RL suffers if the action space or state space is too large. Clearly, policy-based RL is more appropriate for our problem, and hence we elaborate on it in the next section.

B. Policy Gradient Method

The policy gradient method avoids the problem in value-based RL by not estimating the Q values but instead working on the policy directly. Essentially, it improves[2] the policy iteratively until convergence. Moreover, policy gradient methods do not calculate rewards but use probability to select actions, thus they do not need to maintain a status table for calculating rewards.

[2]One policy π_t improves another policy π_{t-1} if at each state the value of action selected by π_t is no smaller than the value of action selected by π_{t-1}.

Algorithm 1 Policy Gradient

```
function REINFORCE
    Initialize θ arbitrary
    for each episode s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T ~ π_θ
        for t=1 to T-1 do
        θ ← θ + α∇_θ log π_θ(s_t, a_t) v_t
        endfor
    endfor
    Return θ
end function
```

The representative method of policy-based RL, REINFORCE, is shown in Algorithm 1. It trains a probability distribution through policy sampling and enhances the probability of selecting actions that lead to high returns. In other words, it directly strengthens the probability of selecting good behaviors and weakens the possibility of selecting bad behaviors via rewards. In policy gradient, the policy π_θ is generally parameterized by neural network parameter θ. Searching for θ that maximizes the expectation of cumulative return $E[\sum_{t=0}^{H} R(s_t)|\pi_\theta]$ normally involves two steps.

1) *Calculate Policy Gradient:*

$$g = q_\pi(s_t, a_t)\nabla_\theta \sum_{t=0}^{T-1} \log \pi(a_t|s_t;\theta) \tag{1}$$

where $q_\pi(s_t, a_t)$ is the score function based on policy π_θ and $\pi(a_t|s_t;\theta)$ is the probability of selecting action a at state s under the current policy.

2) *Use (1) to Adjust and Optimize the Policy Parameter:*

$$\theta = \theta + \alpha g. \tag{2}$$

Intuitively, the higher the score function $q_\pi(s_t, a_t)$ obtains, the more valuable the trajectory is. Hence, after learning enough tracks, the algorithm maximizes the probability of selecting a trajectory that has the highest cumulative reward. It is possible that the cumulative reward value of each trajectory is always positive, in which case no matter what action is taken, the probability of that action will increase. A common way to avoid this situation is to minus a value b, which is called a baseline. Usually, $b = E_\pi[R_t|S = s_t]$. Therefore, in our deep RL approach, we integrate REINFORCE with a baseline algorithm in which the policy gradient is represented as follows:

$$g = \nabla_\theta \sum_{t=0}^{T-1} \log \pi(a_t|s_t;\theta)(q_\pi(s_t, a_t) - b). \tag{3}$$

C. Imitation Learning

Imitation learning is the process in which the agent is given prior information about the environment. Note that such prior information is not considered the "true" model of the environment, but instead the example behavior of expert when facing a similar situation. The expert could be a human or some known optimization algorithms. The agent then tries to

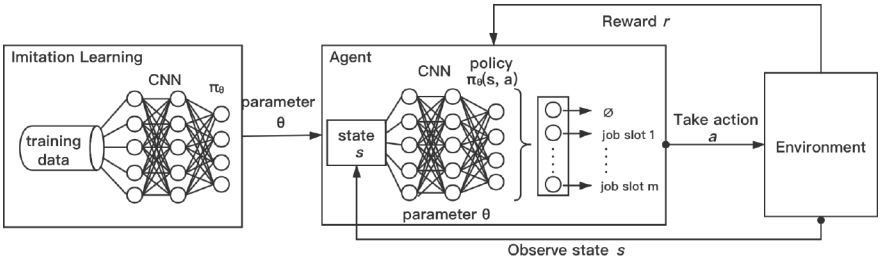


Fig. 1. Overall workflow of DeepRM_Plus.

learn the optimal policy by imitating the expert's decisions. Intuitively, the expert's decisions provide the agent with the right search directions for good solutions, and as such, the agent can learn much faster [36]. Imitation learning is also very helpful in a situation where demonstrating the right behavior is much simpler than calculating the reward function (e.g., teaching a self-driving vehicle).

In [35], imitation learning is used to reduce the exploration space by learning behavioral strategies from expert trajectories. Expert decisions contain a sequence of states and actions. We extract all the state–action pairs to construct a new set, where the state is taken as a feature and the action is taken as a label for classification (for discrete action) or regression (for continuous action), so as to obtain the optimal strategy model. The goal of imitation learning is to generate a set of state–action pairs, whose distribution matches that of the expert's trajectory.

The mainstream methods of imitation learning fall into two categories: 1) behavior cloning and 2) inverse imitation learning. In our research, we adopt behavior cloning in which an agent extracts the behavior trajectory of the expert $\gamma = (s_1, a_1, s_2, a_2, \ldots, \ldots, s_n, a_n)$ by repeatedly studying the behavior of the expert. The agent checks the saved expert tracks to match the current situation it encounters and then imitates the actions taken by the expert. We will further explain the meaning of the expert in our context in Section IV-C.

D. System Assumptions and Problem Formulation

Resource Model: Assume that a data center hosts d types of resources. To ease explanation, we assume that *conceptually* each resource type is organized as a (resource) cluster of capacity $c_i (i = 1, \ldots, d)$. Note that in this article, we only consider resources that are directly relevant to executing a task, e.g., CPU and memory. We do not consider the networking components, such as routers and switches.

Jobs: Each job J_i is denoted by a tuple $J_i = <r_i^1, r_i^2, \ldots, r_i^d>$, where $r_i^j \geq 0 (j = 1, \ldots, d)$ indicates the demand of J_i with respect to resource type j. We assume that the jobs arrive randomly to the system with an arrival rate of λ [37].

Scheduling Decisions: We assume that all the jobs are non-preemptive. A job is considered finished if and only if all its required resources are satisfied. Assume that time is slotted in the unit of times. A scheduling decision for job J_i at time t is denoted by $\mathcal{D}_i(t) = <x_i^1(t), x_i^2(t), \ldots, x_i^d(t)>$, where $x_i^j(t) (\geq 0)$ represents the amount of type j resource allocated to J_i. J_i runs at time t as long as it uses any of the d resources at time t.

Objectives: We define the *cycling time* of J_i as $T_i = T_{ei} - T_{si}$, where T_{ei} is the finish time of J_i and T_{si} is the time when J_i arrives at the data center. We define the *weighted turnaround time* of J_i as $S_i = T_i / T_i^e$, where T_i^e is the actual execution time of job i. In this article, within the same RL framework, we investigate two optimal resource scheduling problems.

Problem 1 (Scheduling for Minimum Weighted Turnaround Time):

$$\underset{\mathcal{D}_i(t)}{\text{minimize}} \lim_{n,t\to\infty} \sum_{i=1}^{n} S_i$$

$$\text{subject to} \sum_{i=1}^{n} x_i^j(t) \leq c_i \quad \forall t, j = 1, \ldots, d.$$

Problem 2 (Scheduling for Minimum Cycling Time):

$$\underset{\mathcal{D}_i(t)}{\text{minimize}} \lim_{n,t\to\infty} \sum_{i=1}^{n} T_i$$

$$\text{subject to} \sum_{i=1}^{n} x_i^j(t) \leq c_i \quad \forall t, j = 1, \ldots, d.$$

Remark 1: Scheduling algorithms could be designed at different abstraction layers. Compared to the work that considers detailed geographically distant data centers, our work is at a higher abstraction layer in the sense that we combine the resources of multiple servers (even if these servers are geographically distant) and regard them as logically clustered resources. At this abstraction layer, we only need to consider the requirements of each job and the (logical) cluster resource utilization when scheduling jobs. In real-world systems, underneath our task scheduling, there should be a lower layer scheduler that maps the logical cluster resources to concrete physical resources (i.e., servers). The lower layer scheduling is beyond the focus of this article.

IV. Details of DeepRM_Plus

A. Workflow of DeepRM_Plus

We solve the above optimal resource scheduling problems within the RL framework. Fig. 1 gives the overall flow diagram of DeepRM_Plus. The CNN accepts the image of the current resource utilization in the cluster as input and finally outputs the optimal policy. To reduce the size of the policy search space, we first utilize imitation learning to get expert strategies. Based on the policy obtained from imitation learning, we train

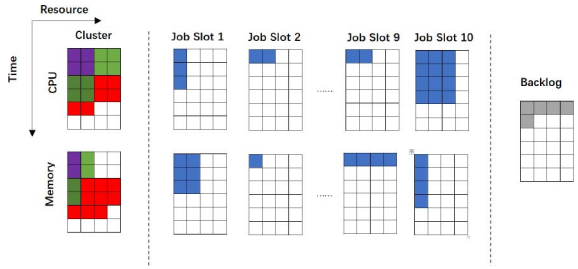


Fig. 2. Presentation of state space.

the neural network via RL until no improvement can be made. Eventually, we get the optimal policy.

B. RL Formulation

To cast the resource scheduling problem as an RL problem, we define the key components as follows.

Environment: It is responsible for generating workload, initializing system (i.e., the resource number, time step into future, waiting queue size, and backlog size) and providing the current state information to the agent (i.e., the next state, jobs left in the system, and the reward for the current action).

State Space: We input the information of the state space to the deep neural network in the form of an image. This way of inputting information to a neural network is based on the great success of the deep neural network in image processing and has been used in DeepRM [17] as well. As shown in Fig. 2, the state of the environment is composed of three parts: 1) (virtual) data center cluster (the leftmost part); 2) waiting queues (the middle part); and 3) backlog queue (the rightmost block).

1) *Leftmost Part:* It represents the current resource allocation. We use CPU (top) and memory (bottom) resources as an example. The horizontal axis of the image indicates the amount of resources, and the vertical axis indicates the next few time slots. Different colors denote different jobs that have been allocated with the resources. In the example of Fig. 2, at the first time slot, the purple job receives 2 units of CPU and 1 unit of memory, and the green job also receives 2 units of CPU and 1 unit of memory. The resource allocation remains unchanged at the second time slot. In the third time slot, the green job receives 2 units of CPU and 1 unit of memory, and the red job receives 2 units of CPU and 3 units of memory.
2) *Middle Part:* It represents the waiting queue that stores jobs waiting for resource allocation. The horizontal length represents the demand for a certain job, and the vertical length represents its ideal completion time. The image of the job slot shows the availability of the waiting queue, and we can get detailed information on the resource requirements for each job by looking at the blue squares in the job slots. In the example of Fig. 2, each job slot in the waiting queue has been occupied. For instance, the job in the first slot requires 1 unit CPU, 2 units memory, and lasts for three time slots.
3) *Rightmost Part:* It represents the backlog queue. The rows are set to the maximum duration of the job and the columns are determined by the maximum number of jobs in each job set. Different from the waiting queue, the backlog queue only shows the number of waiting jobs, not the specific resource requirements of each job.

Action Space: The action represents the behavior of the agent that selects any number of jobs from the waiting queue and allocates the demanded resources for execution under the current state. The action space is defined as $a_t \in A = \{\varnothing, 1, 2, \ldots, M\}$, where M refers to the total number of job slots in the waiting queue. The benefit of this design is that the action space just grows linearly with the number of job slots. Each action means that the agent removes a job slot from the waiting queue (i.e., the middle part of Fig. 2) and allocates the required resources (if possible) for execution (i.e., the leftmost part of Fig. 2). Besides, if the backlog queue holds any extra jobs (i.e., the rightmost part of Fig. 2), remove the first one and put it in the job slot just vacated. $\varnothing$ means that no waiting job is selected, i.e., no change in resource allocation at this time slot.

Reward: The reward is a feedback signal provided by the environment to the agent, whose main purpose is to evaluate how well the agent performs the current action in the current state. Therefore, we use a reward function to guide the agent toward the optimal solution to our goal.

1) The reward function for solving Problem 1 is

$$R_1 = -\sum_{j \in J} \frac{1}{T_{pj}}. \tag{4}$$

We set each reward to a negative value to guide the system to assign jobs as quickly as possible. To minimize the average weighted turnaround time, the reward prefers short jobs since short jobs have more impact than long jobs on $-(1/T_p)$.

2) The reward function for solving Problem 2 is

$$R_2 = -|J| \tag{5}$$

where J is the collection of incomplete assignments, which includes the jobs in execution, jobs in the waiting queue, and jobs in the backlog queue. The reward only relates to the number of unfinished jobs. Therefore, we can punish the agent by counting the number of uncompleted jobs to ultimately minimize the average cycling time.

C. Deep Reinforcement Learning Training Algorithms

As shown in Fig. 1, DeepRM_Plus includes two training stages: 1) imitation learning and 2) online multidimensional resource scheduling by deep RL. Given an image representing the system state, the final output after the two training processes is the optimal policy for resource allocation.

Imitation Learning With RL: The ε-greedy strategy is widely used in both RL and deep RL, to make a balance between exploration and exploitation. Nevertheless, exploration by random actions may lead to a huge exploration space. As a consequence, appropriate decisions may be learned only after many steps based on cumulative rewards. In light of this, we innovatively combine imitation learning with RL to make exploration more efficient. To be more specific, we use behavior cloning to imitate the decisions made by an expert, which

TABLE I
PARAMETERS OF CNN WITH RL

Input	Layer	Filter	Filter Num	Step	Activation Function	Output
20*224*1	Conv1	5*5*1	8	1	ReLu	20*224*8
20*224*8	AvgPool1	2*2	-	2	-	10*112*8
10*112*8	Conv2	5*5*8	16	1	ReLu	10*112*16
10*112*16	AvgPool2	2*2	-	2	-	5*56*16
5*56*16	FC1	-	-	-	tanh	69
69	FC2	-	-	-	Softmax	11

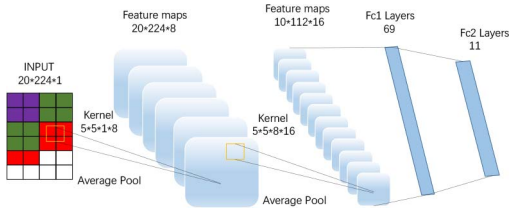


Fig. 3. Architecture of CNN.

are used to guide the search direction of the agent. The behavior cloning algorithm is stated in Algorithm 2.

Remark 2: Our purpose is not to exactly use the expert strategy. Rather, the expert behavior is only a reference for guiding the agent to search in a smaller action space that is likely to lead to the optimal policy. Intuitively, a heuristic that has proven to be effective for a given scheduling objective can be considered as the expert. We will illustrate the impact of the expert in the later evaluation section.

For imitation learning, we need to choose a scheduling algorithm to be imitated based on the predefined goal. For example, to minimize the weighted average time, the SJF algorithm can serve as a candidate expert. We use this strategy to schedule randomly generated n groups of job sets, where each of them contains a different number of jobs. The scheduling results of those jobs are saved as expert decision data.

Putting all the data together, we have a series of expert decision data $\{\tau_1, \tau_2, \ldots, \tau_n\}$, where each decision trajectory $\tau_i (i = 1, 2, \ldots, n)$ contains state–action pairs, denoted as

$$\tau_i = \left\{ s_1^i, a_1^i, s_2^i, a_2^i, \ldots, s_{pi}^i, a_{pi}^i, s_{pi+1}^i \right\} \tag{6}$$

where s_{pi}^i is the state and a_{pi}^i is the corresponding action.

Then, we extract the state–action pairs from the decision data to form a new data set, shown as follows:

$$D = \left\{ \left(s_1^1, a_1^1\right), \left(s_2^1, a_2^1\right), \ldots, \left(s_{p1}^1, a_{p1}^1\right) \left(s_1^2, a_1^2\right), \ldots, \left(s_{p2}^2, a_{p2}^2\right), \ldots, \left(s_{pn}^n, a_{pn}^n\right) \right\}. \tag{7}$$

We then treat states in D as samples and actions as labels for supervised learning. We divide D into two groups[3] (the training set and the test set). The data are then used as input to train the neural network (i.e., lines 13–22 in Algorithm 2).

[3]We split it into training set and test set with a ratio of 9:1 in our later experiment.

Algorithm 2 Cloning Algorithm With RL

Input:
n random job sets $J_1, J_2, ..., J_n$
"Expert" policy π_E
Output:
Imitation Learning parameter θ
1: Set minimum standard accuracy of training set *accu*
2: Create (*state*, *action*) data set D
3: Initialize neural network parameter θ
4: For each $J_i, i = 1, \ldots, n$, set running times T_i for the jobs in J_i.
5: **for** each $J_i, i = 1, \ldots, n$ **do**
6: For each job j in J_i, find its corresponding running time in T_i
7: Get the current cluster state s_j
8: Get the action a_j taken under state s_j by π_E
9: Save (s_j, a_j) in D as a set of samples and labels
10: $s_{j+1} \leftarrow s_j$
11: **end for**
12: Divide D into training set D_1 and D_2
13: **for** each iteration **do**
14: for (s_i, a_i) in D_1 do
15: Using s_i, a_i to train neural network θ
16: end for
17: Calculate $accu_\theta$ in D_2 under the strategy π_θ
18: **if** $accu_\theta > accu$ **then**
19: Save the current value of θ
20: Break;
21: **end if**
22: **end for**

Remark 3: Equation (6) denotes the state–action trajectory when the expert schedules the ith job set, where pi is the total number of state–action pairs in the ith job set. Since the expert schedule is deterministic, the mapping from the state to action is unique. The redundant (s, a) pairs recorded in (7), if any, should be removed before using them to train the neural network.

Convolutional Neural Network With RL: Since we use images to represent the state space (shown in Fig. 2), we input the images into a CNN of six layers, including two convolutional layers of eight kernels and sixteen kernels, respectively, two average pooling layers, and two fully connected layers, as shown in Fig. 3. The output from the last layer is the probabilities of taking actions under the current state. The parameters of CNNs are listed in Table I.

Overall Training Algorithm: After the process of behavior cloning, our system avoids blind exploration in the initial period of RL. With imitation learning, the way of expert decision making is used as a substitute for the traditional random searching. Algorithm 3 shows the whole process.

First, we initialize the neural network of deep RL with parameters trained in imitation learning. Then, we generate a series of jobs and schedule them to get the trajectory τ_i^m according to the strategy of deep RL. Next, we calculate the score function R_t^m at each moment according to the state, the action, and the reward recorded. The baseline of each moment b_t, as introduced in Section III-B, is computed to standardize the score function and reduce the variance of the policy gradient. Finally, the difference between reward R_t^m and baseline b_t is used to update the model parameter. Specifically, the parameter is updated by (3).

D. Analysis of DeepRM_Plus

Since DeepRM_Plus is a type of deep learning algorithm, convergence is an important factor showing the reliability of the algorithm. Generally speaking, backpropagation is the most efficient way to train a neural network. By adjusting the parameters according to the error between the estimated value and the actual value, the algorithm will eventually converge after iterations. Thus, the first stage of DeepRM_Plus, represented by Algorithm 2, will converge to an optimal value in the end.

In the second stage of DeepRM_Plus, as shown in Algorithm 3, a Monte Carlo policy gradient method is applied. Policy gradient is a type of policy iteration that contains policy evaluation and policy improvement. Next, we will show the convergence of the two steps.

1) *Convergence of Policy Evaluation:* Policy evaluation is the process that estimates state value function v_π. According to the Bellman Equation, v_π can be computed as follows:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p\big(s', r|s, a\big)\big[r + \gamma v_\pi(s')\big]. \quad (8)$$

We can present v_π in the following matrix form:

$$V_\pi = R_\pi + \lambda P_\pi V_\pi \quad (9)$$

where R_π, V_π, and P_π can be denoted respectively, by

$$R_\pi = \{R_0, R_1, \ldots, R_n\}^T \quad (10)$$

$$V_\pi = \{V_\pi(s_0), V_\pi(s_1), \ldots, V_\pi(s_n)\}^T \quad (11)$$

$$P_\pi = \begin{bmatrix} 0, & \sum_a \pi(a|s_0)p(s_1, r|s_0, a) & \ldots, & \sum_a \pi(a|s_0)p(s_n, r|s_0, a) \\ \sum_a \pi(a|s_1)p(s_0, r|s_1, a) & 0 & \ldots, & \sum_a \pi(a|s_1)p(s_n, r|s_1, a) \\ , \ldots, & , \ldots, & \ldots, & \cdots \\ \sum_a \pi(a|s_n)p(s_0, r|s_n, a) & \sum_a \pi(a|s_n)p(s_1, r|s_n, a) & \ldots, & 0 \end{bmatrix}. \quad (12)$$

Equation (9) shows that V_π is the fixed point of function $F(x)$ [e.g., $F(x)$ is a contracting mapping], where $F(x) = x$. Then, we randomly initialize x_0 and evaluate $F(x)$ recursively.

a) $n = 1$, $x_1 = F(x_0)$.
b) $n = 2$, $x_2 = F(x_1) = F(F(x_0))$.
c) $n = 3$, $x_3 = F(x_2) = F(F(x_1)) = F(F(F(x_0)))$.

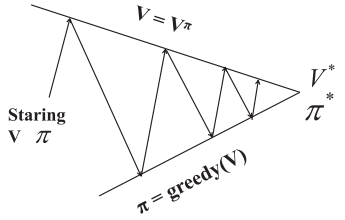


Fig. 4. Policy improvement [38].

According to the *contracting mapping theorem*, $\lim_{n\to\infty} F^n(x_0) = x_*$ where x_* is the state value V_π we want to find. Therefore $\forall\ x_0$, $F(x) = R_\pi + \lambda P_\pi x$ exists a fixed point V_π. Thus, policy evaluation will converge.

2) *Convergence of Policy Improvement:* Policy improvement goes through each state and each action. It calculates $q_\pi(s, a)$ for each action. The current policy is an action with the maximum $q_\pi(s, a)$, i.e., $\pi(a|s)$ is the distribution of an action corresponding to state s, which satisfies $\pi(a|s) = argmax q_\pi(s, a)$. Due to the way of choosing an action, for each state s, $\pi'(s) = \text{argmax}_a\ q_\pi(s, a)$ are satisfied, where $\pi'(s)$ is the updated policy. Once a policy π has been improved using v_π to yield a better policy π', we can then compute $v_{\pi'}$ and improve it again to get a better policy π''. Repeat the process until $\pi'(s) = \pi(s)$. At this time, $\text{argmax}_a\ q_\pi(s, a) = \pi(s)$ and therefore the policy improvement is converged. This typical process is shown in Fig. 4.

Similarly, the policy gradient converges to an optimal policy $\pi_{\theta*}$ after finitely repeating two steps.

1) Policy evaluation, which estimates the score function $q_\pi(s_t, a_t)$ for the current policy π_θ.
2) Policy improvement, which uses $q_\pi(s_t, a_t)$ to get improved policy $\pi_{\theta'}$.

Moreover, policy evaluation itself is an iterative computation leading to a great increase in the speed of convergence of the policy gradient. SJF, which is used as an expert in Algorithm 2 has the time complexity of $O(n \log n)$. Therefore, DeepRM_Plus not only converges fast but with low complexity.

V. Experimental Evaluation

A. Experimental Settings

The proposed work is experimented on Tensorflow. The simulation scenarios are written in Python. The experiment is launched in a desktop computer with hardware configuration as follows.

1) *CPU:* AMD Ryzen 5 1600, 3.2 GHz, 6 cores, 16 GB.
2) *SSD:* GALAX TA1D0120A 120 GB.
3) *GPU:* GTX 1060, 6-GB GDDR5.
4) *Operation System:* Ubuntu 16.04 LTS.

We first carry out the experiment with the total resource amount as 10 units of CPUs and 10 units of memory.[4] The cluster can reserve resources up to 20 time slots in advance, i.e., the cluster diagram at the leftmost part of Fig. 2 is

[4]The unit is an abstract measure for resource allocation granularity. For instance, 1 unit memory could be 100 GB in size, depending on the underline lower layer mapping between logical resources and physical resources.

Algorithm 3 Training in DeepRM_Plus

Input:
n random job sets $J_1, J_2, ..., J_n$
Neural network parameter θ^* trained by imitation learning
Output:
Neural network parameter θ trained by deep reinforcement learning

Initialize neural network model by the parameter of imitation learning:$\theta \leftarrow \theta^*$
For each $J_i, i = 1, \ldots, n$, set running times T_i for the jobs in J_i.
for each iteration **do**
  $\Delta\theta \leftarrow 0$
  for each $J_i, i = 1, \ldots, n$ **do**
    Repeat M times to record M tracks
    for each track $M_m, m = 1, \ldots, M$ **do**
      Get the trajectory

$$\tau_i^m = \{s_1^m, a_1^m, r_1^m, ..., s_{pi}^m, a_{pi}^m, r_{pi}^m\} \sim \pi_{\theta^*}$$

      For each moment t in T_i, calculate the score function $R_t^m = \sum_{s=t}^{|T_i|} \gamma^{s-t} r_s^m$
    end for
    For each moment t in T_i, calculate the baseline $b_t = \frac{1}{M}\sum_1^M R_t^m$
    for each track $M_m, m = 1, \ldots, M$ **do**
      Calculate the gradient of θ:

$$g = \nabla_\theta \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t;\theta)\left(R_t^m - b_t\right)$$

      $\Delta\theta \leftarrow \Delta\theta + \alpha g$
    end for
  end for
  Update the neural network parameter from all scheduling trajectories: $\theta \leftarrow \theta + \Delta\theta$
end for

TABLE II
COMPARED ALGORITHMS

Algorithm	Description
Random	Random scheduling of jobs without priority
Heuristic algorithms	FCFS (first come first service), SJF (Shortest Job First), HRRN (Highest Response Ratio Next)
Tetris [4]	A cluster scheduling algorithm proposed by Microsoft, which matches the multi-resource task requirements to the resource availability of the machine
DeepRM [17]	A representative deep RL-based resource scheduling method

made up of two grids with 20 rows and 10 columns each. In addition, the waiting queue consists of 10 job slots and the maximum number of backup logs is set to 60. These are default settings, and we will change these parameters to test the performance under different cluster configurations in Section V-C3.

We compare DeepRM_Plus with the following baselines: random scheduling, heuristic algorithms, Tetris, and DeepRM, as listed in Table II.

B. Design of Workload

The jobs to be scheduled in the system are divided into 25 sets and the number of jobs in each set is between 5 and 50. We assume that all jobs arrive randomly, i.e., the Poisson arrivals, at a mean rate λ. Such a model has been shown to be effective in modeling data center job arrivals [37]. By adjusting the parameter λ, the average load changes between 10% and 190% of the cluster capacity.

We tested a similar workload setting in [17]. The cluster resources include CPU and memory with capacity $\{1r, 1r\}$, respectively, where $r = 10$ as stated in Section V-A. The resource requirements for each job are as follows: each job randomly selects one resource as the primary resource and the other resource as the auxiliary resource. The demand for the primary resource is random between $0.5r$ and $1r$, while the demand for the auxiliary resource is random between $0.1r$ and $0.2r$. As for the duration of each job, 80% of the jobs are short-term jobs with a duration of 1–3 time slots and the rests are long-term jobs with a duration of 10–15 time slots.

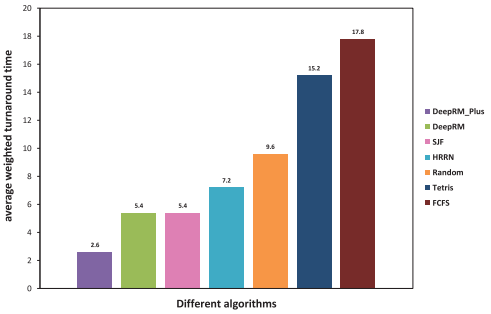


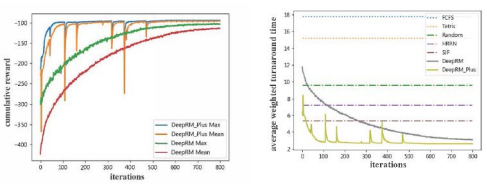
Fig. 5. Comparison of average weighted turnaround time.

C. Performance Comparison

In this section, we carry out three sets of experiments. The first set of experiments aims to show the performance of DeepRM_Plus with different goals. The second set of experiments is to demonstrate the performance of DeepRM_Plus under different cluster configurations. The last set of experiments aims to evaluate the performance of DeepRM_Plus under Problem 1 in comparison to baseline algorithms using the real-world *Alibaba-Cluster-trace-v2018* trace data [39].

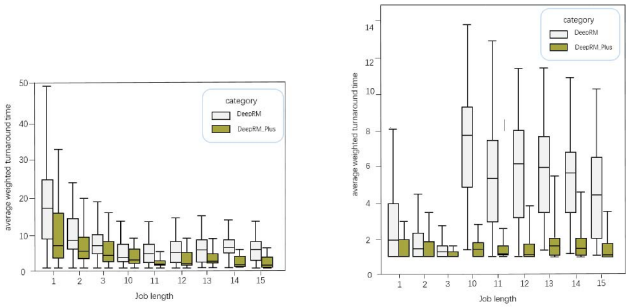
1) Minimize Average Weighted Turnaround Time (Problem 1): In this case, DeepRM_Plus uses R_1 as the reward function. We investigate the training process with a cluster load of 140% ($\lambda = 0.7$). Fig. 5 shows the overall comparison of different scheduling methods with respect to the average weighted turnaround time. We can see that DeepRM_plus achieves the best performance. Compared to DeepRM, DeepRM_plus reduces average weighted turnaround time by 51.85%.

To see why DeepRM_plus is better than DeepRM, we compare the two methods in terms of the convergence rate and the evolution of average weighted turnaround time, which are demonstrated in Fig. 6. In RL, the main goal is to maximize the cumulative reward of a certain trajectory. As shown in Fig. 6(a), the cumulative rewards for both two algorithms increase over iterations and the training time of each epoch of DeepRM_Plus is 37.5% less than that of DeepRM. Moreover, Fig. 6(a) depicts the maximum and average cumulative reward



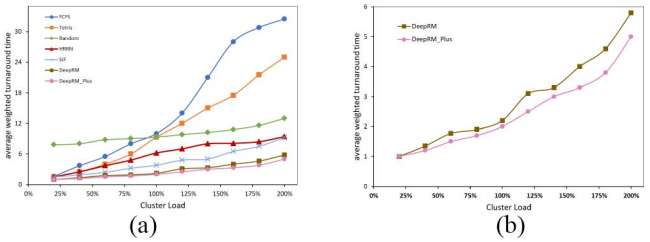
(a) (b)

Fig. 6. Training process. (a) Cumulative rewards. (b) Iterative training.



(a) (b)

Fig. 7. Average weighted turnover time versus job duration. (a) Initial stages (< 800 iterations). (b) After 800 iterations.



(a) (b)

Fig. 8. Average weighted turnaround time under different load. (a) Different algorithms. (b) DeepRM and DeepRM_Plus.

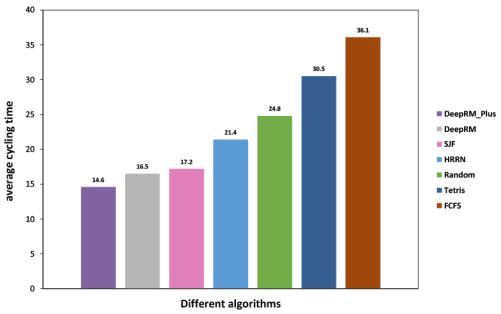


Fig. 9. Comparison of average cycling time.

values of DeepRM_Plus and DeepRM algorithms in each iteration. For both algorithms, the gap between the maximum cumulative reward and the average cumulative reward represents whether further training is needed. If the decisions of some samples are far better than average actions, then the current strategy may still have room for further improvement. Otherwise, the model has stabilized as the gap becomes small.

Fig. 6(b) demonstrates the results of the average weighted turnaround time for different scheduling algorithms. We can see that SJF holds the minimum value of the average weighted turnaround time among those heuristic algorithms. It means that SJF is the best expert for an agent to imitate. Hence, in the following experiments, we regard SJF as the expert to find the optimal policy. This imitation learning helps DeepRM_Plus converge faster than DeepRM. Although there are some oscillations during initial iterations (< 500), the agent gradually learns to stabilize quite well after 500 iterations, transcending all benchmark models during and after their learning saturation points.

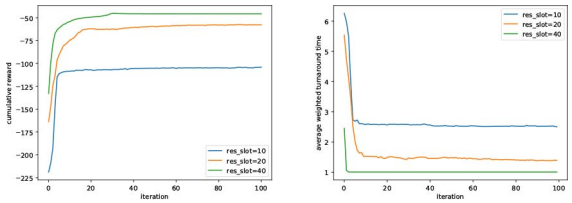
To further investigate how deep RL reduces the average weighted turnaround time, we draw box plots for each job length before and after training. Fig. 7(a) and (b) shows the scheduling results of all jobs when DeepRM_Plus and DeepRM are initializing and after 800 iterations of training, respectively. In the initial stage, the scheduling process is similar to the random algorithm where the job selection is independent of job duration. When the cluster is full, both short and long jobs will wait. According to the definition of weighted turnaround time, when a short job and a long job waits for the same amount of time, this metric has a larger increase for the short job than for the long job. Therefore, in the beginning, the weighted turnaround time of short jobs is higher. Both deep RL algorithms have a large weighted turnaround time since the number of short jobs is three times more than the number of long jobs. With the aid of the reward function and iterative training, the system realizes that reducing the weighted turnaround time of short jobs and appropriately increasing the weighted turnaround time of long jobs are more conducive to reducing the average weighted turnaround time of all jobs.

Finally, we test DeepRM_Plus with different cluster loads. The test results are displayed in Fig. 8(a). We can see that the average weighted turnaround time of each algorithm is proportional to the cluster load. The average weighted turnaround time of FCFS rises the fastest while the curve trend of DeepRM_Plus is the smoothest. When the cluster load is over 60%, DeepRM and DeepRM_Plus both outperform other algorithms. Note that among all other heuristic algorithms, SJF reduces the weighted turnaround time to the maximum extent. This again shows that SJF is a good expert to help the agent search more effectively.

We further compare the two deep RL methods. As shown in Fig. 8(b), DeepRM_Plus surpasses DeepRM when the load is higher than 40%. Moreover, when the cluster load becomes higher, the advantage of DeepRM_Plus is more obvious.

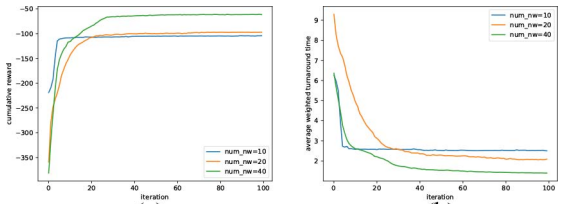
2) Minimize Average Cycling Time (Problem 2): For this goal, we use reward function R_2 while keeping the same structure of the neural network. Since we have analyzed the training process and the mechanism of DeepRM_Plus at length in the last part, to make this article concise, we only show the sample final results at the workload of 140% cluster capacity. As shown in Fig. 9, DeepRM_Plus beats DeepRM by 11.51% reduction in the average cycling time. In addition, DeepRM_Plus performs the best among all the evaluated baseline algorithms.

3) DeepRM_Plus With Different Cluster Configurations: In the second set of experiments, we conduct the following experiments with different cluster configurations and the reward function defined by (4). We plot the performance of DeepRM_Plus with different cluster configurations in Fig. 10.



(a) (b)

Fig. 10. Performance comparison when num_nw = 10. (a) Reward under different res_slot. (b) Average weighted turnaround time under different res_slot.



(a) (b)

Fig. 11. Performance comparison when res_slot = 10. (a) Reward under different num_nw. (b) Average weighted turnaround time under different num_nw.

The number of CPU and memory denoted as *res_slot* influences the scheduling process. First, we choose the value of *res_slot* from {10, 20, 40} to run the experiment. The result in Fig. 10 shows that DeepRM_Plus tends to create a higher total reward and a lower average weighted turnaround time as the amount of resources increases. This indicates that with more resources, DeepRM_plus can achieve better performance.

Apart from *res_slot*, we also change the size of the waiting queue, represented as *num_nw*, to test the performance. We set the waiting queue to be {10, 20, 40} and record the scheduling results accordingly in Fig. 11. Fig. 11(a) demonstrates that the cumulative reward converges to a higher value when the waiting queue becomes larger. Note that the growth of waiting time for a job increases its cycling time (i.e., T_{pj}). Thus, the cumulative reward will increase according to (4).

From Fig. 11(b), we observe that with the increase of *num_nw* value, the average weighted turnaround time decreases. The reason is that as the waiting queue becomes larger, more short jobs will join in the waiting queue and will be executed first. Although the long jobs will wait for a longer time based on our policy, the increased weighted turnaround time for long jobs is smaller than the decreased weighted turnaround time for short jobs. As for the backlog size, it can be just slightly larger than the number of jobs in the job set. Otherwise, it will lead to a sparse state space, which makes the feature extraction of the state space difficult.

4) Trace Driven Simulation for DeepRM_Plus: In the third set of experiments, we conduct trace-driven simulation based on *Alibaba-Cluster-trace-v2018* data, which records the workloads of the cloud cluster at a different time of a day. The trace data contain a large number of jobs over 4000 machines in a period of eight days. We first analyze the feature of the trace data, which is demonstrated in Fig. 12. It shows that long jobs account for a large proportion. We randomly choose 20 000 tasks and perform an evaluation on them. Each record in the data contains four attributes relevant to our simulation: 1) start time; 2) end time; 3) cpu requirement; and 4) memory

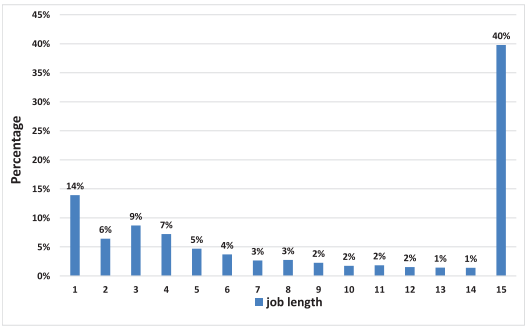
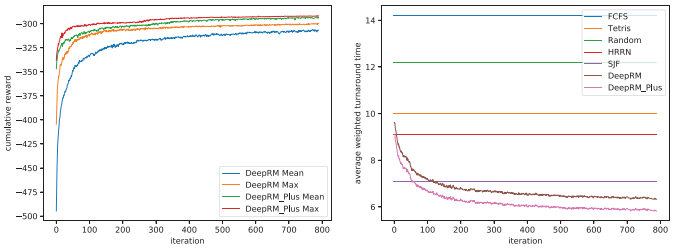


Fig. 12. Distribution of job length for the Alibaba cluster data.



(a) (b)

Fig. 13. Performance comparison among algorithms for the Alibaba cluster data. (a) Cumulative reward for Alibaba-cluster-trace. (b) Average weighted turnaround time for Alibaba-cluster-trac.

requirement. To align with our experiment, we preprocessed the data to have the same input format as DeepRM_Plus.

We use (4) as the reward function to run the simulations. With the same baseline algorithms introduced in the previous sections, we report the cumulative reward and the average weighted turnaround time of DeepRM_Plus over those algorithms in Fig. 13. As shown in Fig. 13(a), DeepRM_Plus outperforms DeepRM with respect to the final reward and the convergence rate. In terms of the average weighted turnaround time, Fig. 13(b) shows that DeepRM_Plus exceeds all other algorithms and converges to the lowest value. Note that both the cumulative reward and the average turnaround time are larger than those obtained with simulated data because the distribution of the two kinds of data is different—the majority of jobs are short in the simulated data while the opposite situation occurs in *Alibaba-Cluster-trace-v2018*. The results further demonstrate that DeepRM_Plus can achieve better performance in different job distributions.

5) Overall Summary: Through various experiments with both simulated data and real-world trace data, we can conclude that DeepRM_Plus performs the best among all the baseline methods in both the average weighted turnaround time and the average cycling time. Compared with the state-of-the-art algorithm DeepRM, DeepRM_Plus is 37.5% faster in terms of the convergence rate. Moreover, DeepRM_Plus reduces the average weighted turnaround time and the average cycling time by 51.85% and 11.51%, respectively.

VI. Conclusion

In this article, we proposed a new cloud resource scheduling approach, DeepRM_Plus, which uses deep RL to tackle the challenging multiresource scheduling problem. With CNN and imitation learning, DeepRM_Plus enables an agent to learn an

optimal policy more efficiently. Various experiments showed that DeepRM_Plus is so far the best online scheduling algorithm. It is not only superior to various heuristic algorithms but also surpasses the state-of-the-art DeepRM. Moreover, with only a slight change in the reward function, DeepRM_Plus can train the optimal scheduling strategy for different objectives. In this sense, DeepRM_Plus may be used as a more general framework that facilitates the development of new and efficient solutions for other online resource scheduling problems.

The most important meaning of DeepRM_Plus is that it demonstrates the great potential of integrating imitation learning and deep RL for solving hard resource scheduling problems. Along this line, we, in the future plan, will explore other RL methods, such as actor–critic [40] and DDPG [40], in the domain of resource management.

ACKNOWLEDGMENT

W. Tian conceived main ideas and supervised the work together with K. Wu. W. Guo developed the formulations and performed experiments together with Y. Ye and L. Xu. All authors discussed the results and contributed to the final manuscripts.

REFERENCES

[1] T. Baker, B. Aldawsari, M. Asim, H. Tawfik, Z. Maamar, and R. Buyya, "Cloud-senergy: A bin-packing based multi-cloud service broker for energy efficient composition and execution of data-intensive applications," *Sustain. Comput. Informat. Syst.*, vol. 19, pp. 242–252, Sep. 2018.

[2] T. Baker, M. Asim, H. Tawfik, B. Aldawsari, and R. Buyya, "An energy-aware service composition algorithm for multiple cloud-based IoT applications," *J. Netw. Comput. Appl.*, vol. 89, pp. 96–108, Jul. 2017.

[3] A. Hazarika, S. Poddar, and H. Rahaman, "Survey on memory management techniques in heterogeneous computing systems," *IET Comput. Digit. Techn.*, vol. 14, no. 2, pp. 47–60, Mar. 2020.

[4] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 455–466, 2014.

[5] H. A. Al-Kashoash, H. Kharrufa, Y. Al-Nidawi, and A. H. Kemp, "Congestion control in wireless sensor and 6LOWPAN networks: Toward the Internet of Things," *Wireless Netw.*, vol. 25, no. 8, pp. 4493–4522, 2019.

[6] H. U. Simon, "On approximate solutions for combinatorial optimization problems," *SIAM J. Discrete Math.*, vol. 3, no. 2, pp. 294–310, 1990.

[7] R. M. Karp and C. H. Papadimitriou, "On linear characterizations of combinatorial optimization problems," *SIAM J. Comput.*, vol. 11, no. 4, pp. 620–632, 1982.

[8] J. F. Martinez and E. Ipek, "Dynamic multicore resource management: A machine learning approach," *IEEE Micro*, vol. 29, no. 5, pp. 8–17, Sep./Oct. 2009.

[9] H. Ye and G. Y. Li, "Deep reinforcement learning for resource allocation in V2V communications," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Kansas City, MO, USA, 2018, pp. 1–6.

[10] Y. Zhang, J. Yao, and H. Guan, "Intelligent cloud resource management with deep reinforcement learning," *IEEE Cloud Comput.*, vol. 4, no. 6, pp. 60–69, Nov./Dec. 2017.

[11] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2011.

[12] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Univ. Cambridge, Cambridge, U.K., 1989.

[13] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[14] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," 2013. [Online]. Available: arXiv:1312.5602.

[15] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[16] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[17] 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.

[18] H. Goudarzi, M. Ghasemazar, and M. Pedram, "SLA-based optimization of power and migration cost in cloud computing," in *Proc. 12th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput. (CCGRID)*, Ottawa, ON, Canada, 2012, pp. 172–179.

[19] A. Beloglazov and R. Buyya, "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers," *Concurrency Comput. Pract. Exp.*, vol. 24, no. 13, pp. 1397–1420, 2012.

[20] A. Beloglazov and R. Buyya, "Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 7, pp. 1366–1379, Jul. 2013.

[21] V. Priya, C. S. Kumar, and R. Kannan, "Resource scheduling algorithm with load balancing for cloud service provisioning," *Appl. Soft Comput.*, vol. 76, pp. 416–424, Mar. 2019.

[22] K. Al Nuaimi, N. Mohamed, M. Al Nuaimi, and J. Al-Jaroodi, "A survey of load balancing in cloud computing: Challenges and algorithms," in *Proc. IEEE 2nd Symp. Netw. Cloud Comput. Appl.*, London, U.K., 2012, pp. 137–142.

[23] C. Assi, S. Ayoubi, S. Sebbah, and K. Shaban, "Towards scalable traffic management in cloud data centers," *IEEE Trans. Commun.*, vol. 62, no. 3, pp. 1033–1045, Mar. 2014.

[24] Z. Zhou, P. Liu, J. Feng, Y. Zhang, S. Mumtaz, and J. Rodriguez, "Computation resource allocation and task assignment optimization in vehicular fog computing: A contract-matching approach," *IEEE Trans. Veh. Technol.*, vol. 68, no. 4, pp. 3113–3125, Apr. 2019.

[25] B. Wang, Y. Song, J. Cao, X. Cui, and L. Zhang, "Improving task scheduling with parallelism awareness in heterogeneous computational environments," *Future Gener. Comput. Syst.*, vol. 94, pp. 419–429, May 2019.

[26] S. Raheja, "An intuitionistic based novel approach to highest response ratio next CPU scheduler," *Intell. Decis. Technol.*, vol. 13, no. 4, pp. 523–536, 2019.

[27] A. Marahatta, Y. Wang, F. Zhang, A. K. Sangaiah, S. K. S. Tyagi, and Z. Liu, "Energy-aware fault-tolerant dynamic task scheduling scheme for virtualized cloud data centers," *Mobile Netw. Appl.*, vol. 24, no. 3, pp. 1063–1077, 2019.

[28] S. G. Domanal, R. M. Guddeti, and R. Buyya, "A hybrid bio-inspired algorithm for scheduling and resource management in cloud environment," *IEEE Trans. Services Comput.*, vol. 13, no. 1, pp. 3–15, Jan./Feb. 2020.

[29] F. Farahnakian, P. Liljeberg, and J. Plosila, "Energy-efficient virtual machines consolidation in cloud data centers using reinforcement learning," in *Proc. IEEE 22nd Euromicro Int. Conf. Parallel Distrib. Netw. Based Process.*, Torino, Italy, 2014, pp. 500–507.

[30] S. Telenyk, E. Zharikov, and O. Rolik, "Modeling of the data center resource management using reinforcement learning," in *Proc. IEEE Int. Sci. Pract. Conf. Problems Infocommun. Sci. Technol. (PIC S&T)*, Kharkiv, Ukraine, 2018, pp. 289–296.

[31] O. Rolik, E. Zharikov, A. Koval, and S. Telenyk, "Dynamie management of data center resources using reinforcement learning," in *Proc. IEEE 14th Int. Conf. Adv. Trends Radioelecrtron. Telecommun. Comput. Eng. (TCSET)*, Slavske, Ukraine, 2018, pp. 237–244.

[32] D. Zeng, L. Gu, S. Pan, J. Cai, and S. Guo, "Resource management at the network edge: A deep reinforcement learning approach," *IEEE Netw.*, vol. 33, no. 3, pp. 26–33, May/Jun. 2019.

[33] B. Dab, N. Aitsaadi, and R. Langar, "Q-learning algorithm for joint computation offloading and resource allocation in edge cloud," in *Proc. IFIP/IEEE Symp. Integr. Netw. Serv. Manag. (IM)*, Arlington, VA, USA, 2019, pp. 45–52.

[34] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *J. Artif. Intell. Res.*, vol. 4, pp. 237–285, May 1996.

[35] B. Price and C. Boutilier, "Accelerating reinforcement learning through implicit imitation," *J. Artif. Intell. Res.*, vol. 19, pp. 569–629, Dec. 2003.

[36] M. Bojarski *et al.*, "End to end learning for self-driving cars," 2016. [Online]. Available: arXiv:1604.07316.

[37] S. Di, D. Kondo, and F. Cappello, "Characterizing and modeling cloud applications/jobs on a Google data center," *J. Supercomput.*, vol. 69, no. 1, pp. 139–160, 2014.

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

[39] *Alibaba-Cluster-Trace-2018*, Alibaba, Hangzhou, China, 2018. [Online]. Available: https://github.com/alibaba/clusterdata/blob/v2018/cluster-trace-v2018/trace_2018.md

[40] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems*. Cambridge, MA, USA: MIT Press, 2000, pp. 1008–1014.